Universidad Carlos III de Madrid

Trabajo Fin de Grado

Planificación de trayectorias para vehículos submarinos con *ROS*

Grado en Ingeniería de Tecnologías Industriales

Departamento de Ingeniería de Sistemas y Automática

Autor:
Ana Laura Bermejo Escudero

Tutor:
Alberto Jardón Huete

2016-2017

Agradecimientos

Parecía que nunca iba a llegar este momento, pero finalmente ha llegado. Con todo el documento escrito es hora de agradecer a todos aquellos que han aportado su granito de arena en el proyecto.

En primer lugar agradecer a mi tutor Alberto la ayuda mostrada durante este tiempo, y a Olaya la paciencia que ha tenido con todas mis dudas, ya que sin ella no hubiese conseguido realizar el trabajo con éxito.

A mi familia, agradecerles todo el apoyo mostrado no solo ahora, sino durante toda la carrera y toda la vida. Gracias a ellos he llegado a donde estoy ahora y soy quién soy.

No puede faltar hueco para mis amigos y compañeros de clase, ya que han conseguido que las vísperas de exámenes o entregas fuesen más divertidas.

A todos, gracias de corazón.

Resumen

El presente documento muestra el Trabajo de Fin de Grado realizado *Planificación de trayectorias para vehículos submarinos con ROS* en la Universidad Carlos III de Madrid, en la Escuela Politécnica Superior de Leganés.

El proyecto desarrollado consta de dos partes bien diferenciadas: un estudio teórico y una simulación. Ambas partes son independientes pero complementarias entre sí.

En la primera parte se ha realizado un estudio teórico de los distintos algoritmos existentes para la planificación de trayectorias de vehículos autónomos, profundizando en aquellos empleados en la simulación, correspondiente a la parte práctica del proyecto.

En esta parte práctica, utilizando el *framework* de desarrollo de robots *MoveIt* y los algoritmos definidos por defecto en este entorno de trabajo, se han planificado trayectorias para el desplazamiento del vehículo autónomo submarino (AUV) GIRONA 500. Este robot pertenece al Instituto de VIsión por COmputador y ROBótica (ViCOROB) de la Universidad de Gerona. Para la visualización de las trayectorias se ha utilizado el entorno de visualización en 3D proporcionado por la herramienta *Rviz*.

Abstract

The current document presents the bachelor thesis *Underwater vehicles motion planning with ROS* for University Carlos III of Madrid on the Polytechnic School of Leganés.

The development of this project has two main areas: a theoretical study and a simulation. Both of them are independent of each other, but complementary between them.

On the first area, a theoretical study has been done in order to study the different algorithms that exists for motion planning, emphasizing on those that are used on the simulation; corresponding to the practical part of the thesis.

In this practical part, using the development framework of robots *MoveIt* and the algorithms already defined on this working space, a motion planning has been done for the Autonomus Underwater Vehicle (AUV) GIRONA 500. This robot belongs to the Institute of Computer Vision and Robotics (ViCOROB), which is in Girona University. In order to visualize the motion, *Rviz* has been used as the 3D visualization environment.

Índice

Índice de figuras

Índice de tablas

1. Introducción

Un robot móvil es una máquina automática que cuenta con la capacidad de moverse en un entorno. Existen numerosos tipos de robots en la práctica: los brazos robóticos, los robots autónomos, los robots móviles, los teleoperados, etc. Aunque los problemas de estos son diferentes, la planificación de trayectorias para todos ellos es prácticamente la misma, ya que el principal objetivo en estos casos es moverse sin colisionar con ningún objeto presente en la trayectoria.

El término planificación de trayectorias se utiliza para guiar un robot de un punto inicial a uno final sin colisionar con ningún objeto que se encuentre en la ruta. Un algoritmo de planificación de trayectorias debe producir, a partir del objetivo que se pretenda alcanzar y de los datos disponibles del entorno en el que se debe llevar a cabo la navegación, una serie de comandos que permitan al robot transitar de forma segura en un entorno con obstáculos hasta completar la tarea.

A lo largo de este trabajo se estudiarán los distintos algoritmos que se suelen emplear para la generación de trayectorias.

La robótica móvil es un campo de estudio que permite explorar entornos inaccesibles al ser humano por su lejanía, coste o peligrosidad y para realizar tareas poco agradables o laboriosas y repetitivas. Aunque el área de la robótica móvil es relativamente nueva, la robótica tiene su origen hace miles de años, ya que el ser humano siempre ha construido máquinas que imitasen partes de su cuerpo, principalmente brazos.

1.1. Motivación

Un buen método de planificación de trayectorias ha de ser capaz de resolver multitud de problemas, independientemente del entorno, las situaciones o el sistema en el que se aplica, ya que no es óptimo variar el planificador en función de la situación o la plataforma. Esta variación conlleva que el sistema sea capaz de detectar las situaciones y variar el planificador de forma transparente, lo cual no solo añade complejidad al problema, sino un tiempo de ejecución que puede ser crítico. Además, en muchas ocasiones los métodos desarrollados solo contemplan el marco teórico, sin tener en cuenta las complicaciones que surgen al adaptar los algoritmos a un entorno real.

Los océanos se extienden por tres cuartas partes de la superficie de nuestro planeta, y sin embargo son las zonas más desconocidas del planeta. Es por ello que los avances tecnológicos se están invirtiendo parte en el desarrollo de submarinos, ya que pueden ser de gran ayuda en exploraciones submarinas donde existe un gran peligro para el ser humano debido a la complejidad del entorno o los anima-

les que habiten en él. Otras aplicaciones serían la búsqueda de restos de naufragios o aviones hundidos en el océano.

El hecho de que estos submarinos sean autónomos facilita mucho la exploración de entornos que se encuentren a una gran profundidad, ya que la conexión con el operario puede ser compleja o situarse a una distancia mayor del operario de la que los sensores pueden alcanzar. Por tanto, si el submarino está dotado de cierta autonomía se podrían llevar a cabo misiones más complejas con éxito.

1.2. Ambientación

Los simuladores de vehículos submarinos se han desarrollado en numerosas ocasiones, pero son en su mayoría desarrollados para un ROV (*Remotely Operated Vehicle*) o un AUV (*Autonomous Underwater Vehicle*) concreto, como por ejemplo el simulador *MORSE* [22], *UWSim* [1], *Gazebo* [23] y *V-REP* [22] entre otros.

Como los AUVs son autónomos, los simuladores son de mayor necesidad, pues es necesario comprobar que el sistema controla correctamente antes de realizar una misión; en cambio los ROV, al ser controlados por un operador en todo momento no existe el mismo riesgo.

Los simuladores son una parte clave de la robótica puesto que permiten depurar los códigos y los modelos matemáticos antes de implementarlos en el robot. Gracias a esto se pueden evitar ciertos errores que pueden provocar que la vida útil del robot quede reducida muy considerablemente.

Otra de las ventajas que presentan los simuladores es facilitar la elección del robot para una determinada tarea sin necesidad de hacer ningún tipo de inversión previa y con unos errores mínimos; como por ejemplo al programar una trayectoria en un robot real, el robot está sometido a agentes externos (como la fricción y más particularmente en el caso de los submarinos a corrientes) y se alcanza el objetivo con un pequeño margen de error. En cambio, si se programa la misma trayectoria en un robot simulado, la probabilidad de acertar incrementa, ya que aunque se simule el roce, este nunca actuará como tal.

En los simuladores se pueden probar códigos y librerías experimentales, que si se ejecutasen directamente en el robot real podrían causar estragos en el mismo.

La importancia de la robótica radica en su amplio impacto en la capacidad de Europa para mantener y ampliar un sector de la producción competitivo. La robótica también ofrece nuevas soluciones a los desafíos de la sociedad desde el envejecimiento a la salud, transporte inteligente, seguridad, energía y medio ambiente.

La comisión Europea promueve activamente la investigación y la innovación mediante robots más seguros, salvaguardando al mismo tiempo los aspectos éticos de los progresos alcanzados. El enfoque de la Comisión europea está en construir sobre un esfuerzo continuo para desarrollar la base científica y así empujar los límites de la tecnología.

La robótica es un mercado en rápido desarrollo cada vez más impulsado por el desarrollo de productos novedosos y mejorados en áreas tan diversas como la fabricación, búsqueda y rescate y recuperación, inspección y monitoreo, cirugía y salud, hogares y automóviles, transporte y logística, agricultura y muchos más [24].

El rápido aumento en el uso de robots en nuestros hogares y en el trabajo, en hospitales y entornos industriales proporciona una visión inspiradora sobre cómo pueden beneficiar a la sociedad en su conjunto y cómo las prioridades para estimular la robótica deben definirse en este momento de su evolución, Mejor desarrollo del potencial de crecimiento, empleo e innovación en Europa.

A través de una cartera de más de ciento veinte proyectos de investigación y acciones de coordinación, la Comisión Europea ha ido construyendo progresivamente una sólida base de intercambio de conocimientos y cooperación en toda la comunidad de actores de la robótica. Esta base ahora incluye una asociación público-privada en robótica llamada SPARC [14].

Este trabajo se lleva a cabo en estrecha colaboración con la comunidad de la robótica, incluidos los programas de los Estados miembros, la industria, las universidades y las instituciones de investigación [87].

1.3. Objetivos

Como se ha comentado anteriormente, este proyecto consta de dos partes, una parte teórica y una simulación de trayectorias. Para poder alcanzar estos objetivos globales se han establecido una serie de objetivos menores que una vez hayan sido cumplidos todos, se podrá considerar que se ha alcanzado aquello que se esperaba del proyecto.

- El objetivo teórico es el estudio y la comprensión de los diferentes algoritmos existentes para la planificación de trayectorias.

- El objetivo principal de la parte de la simulación, es la generación de trayectorias, de manera que el robot deberá ser capaz de alcanzar el objetivo final sin colisionar con ningún objeto que se interponga en su camino. Para poder

lograr este objetivo es necesario establecer objetivos intermedios:

- El completo entendimiento y familiarización con el entorno de *Ubuntu*, puesto que es la primera vez que se utilizará este sistema operativo.

- Estudio y análisis de *ROS* [32]. Se deberá realizar un estudio y análisis de *ROS* para su comprensión. Esto permitirá que el desarrollo del sistema sea más fácil y sencillo. Además de la utilización de algunas herramientas y simuladores de *ROS* como son *MoveIt* [29], *UWSim* [1] y *Rviz* [15]

1.4. Descripción del problema

Un vehículo submarino autónomo es un robot que navega bajo el agua sin necesidad de ningún operario.

En este trabajo se estudiarán los diferentes algoritmos de planificación de trayectorias y se generará una trayectoria para el robot submarino del simulador UWSim (UnderWater Simulator) con la ayuda de la herramienta de visualización 3D *Rviz*.

El UWSim tiene integrados distintos entornos de simulación, que serán estudiados a lo largo de la memoria. Estos entornos pueden ser fácilmente modificados, así como los sensores que se utilizarán para resolver el problema propuesto.

Este simulador se utiliza para la investigación y desarrollo de los robots acuáticos. Fue creado para probar e integrar distintos algoritmos de control para los proyectos de investigación RAUVI [16] y TRIDENT [17].

Las investigaciones de estos proyectos fueron financiadas por el Ministerio de Investigación e Innovación de España.

1.5. Estructura del documento

Este documento se divide en diez capítulos, de los cuales el último alberga los distintos anexos del proyecto. A continuación se realiza una breve descripción de cada uno de ellos:

1. **Introducción:** en esta sección se hace una breve descripción del problema a tratar en este proyecto y los objetivos que se quieren conseguir.

2. **Estado del Arte:** en esta sección del documento se realiza una introducción y puesta en contexto de los vehículos autónomos sumergibles, así como los métodos empleados para resolver la planificación de trayectorias para este tipo de robots.

3. **Hardware:** se especifica el material necesario para el desarrollo del proyecto.

4. **Software:** se profundiza en las distintas herramientas empleadas en la generación de trayectorias para el submarino g500.

5. **Marco regulador:** hace referencia a la legislación de vehículos autónomos sumergibles vigente actualmente en España.

6. **Trabajo experimental:** se describe el proceso llevado a cabo para la generación de trayectorias con el submarino g500.

7. **Resultados:** se hace un pequeño análisis de los resultados obtenidos en la simulación.

8. **Conclusiones y futuros trabajos:** se describen las conclusiones generales obtenidas y se realiza una breve descripción de los posibles trabajos futuros.

9. **Presupuesto y planificación:** incluye el detalle del coste del proyecto y la planificación del tiempo para llevarlo a cabo.

10. **Anexos:** se incluyen los manuales de instalación de las distintas herramientas empleadas en la generación de trayectorias. Además existe un anexo con todos los acrónimos que aparecen a lo largo del proyecto.

2. Estado del Arte

2.1. Submarinos autónomos

En esta sección se realizará una introducción acerca de la evolución de los submarinos autónomos, así como los detalles del submarino que se empleará en la simulación.

Los robots submarinos autónomos se utilizan con un fin científico, comercial y militar, para explorar entornos inaccesibles o peligrosos. También se utilizan para realizar misiones bajo el agua como detectar y ubicar restos de naufragios, ruinas, rocas y en general obstáculos que puedan afectar a la navegación de barcos comerciales.

Los océanos se extienden por tres cuartas partes de la superficie de nuestro planeta, y sin embargo son las zonas más desconocidas de la Tierra. Para solucionar este problema, los avances tecnológicos han permitido desarrollar submarinos autónomos que permiten adentrarse en el océano.

2.1.1. Clasificación de los submarinos

Dado que el objetivo de este proyecto es la planificación de trayectorias con vehículos submarinos, es necesaria una clasificación de los distintos tipos de submarinos. El método más frecuente para clasificarlos es en función de si requieren o no tripulación u operarios.

Los submarinos tripulados son en su mayoría militares, pero también existen no militares,cuya finalidad es apoyar las investigaciones submarinas.

Figura 1: AUV [5]

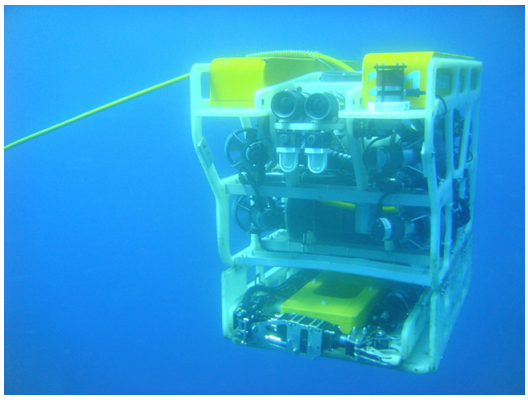


Figura 2: ROV [6]

Los submarinos no tripulados a su vez también se subdividen en clases.

- Los *ROV* son controlados y alimentados desde la superficie por un operador o piloto a través de un cordón umbilical o usando el control remoto ya que carecen de autonomía para realizar las actividades. Algunos *ROV* [10] han sido remolcados detrás de barcos y botes para realizar diferentes tipos de trabajos; la principal operación para estos sistemas es remolcar al vehículo que se encuentra a la profundidad deseada. Los *ROVs* siguen operativos en la actualidad ya que resultan de gran utilidad en determinadas misiones.

- Existen vehículos con capacidad de control sobre sí mismos durante el cumplimiento de una tarea predefinida; éstos son conocidos como *AUVs* (Autonomous Underwater Vehicle).

2.1.2. Los AUVs

Los AUVs son robots que desempeñan tareas debajo del agua. Como se indicó anteriormente, este tipo de vehículos son no tripulados, por tanto son controlados por un ordenador instalado en el dispositivo o también se controlan mediante un operario a bordo del vehículo de apoyo. Los AUVs son accionados por baterías o celdas de combustible y pueden operar al rededor de los 6000 metros de profundidad. Los avances en sistemas de propulsión y fuentes de energía le dan a estos robots mayor duración en cuanto a distancias y tiempos de operación [2].

La tecnología de los AUV ha estado presente desde hace algunas décadas, pero es ahora cuando comienza a ser una opción comercial viable. Últimamente se ha mostrado mucho interés de la industria petrolera y de gas, ya que al emplear este tipo de tecnologías se libran de muchos problemas, costos y riesgos que conllevan las actividades de inspección y muestreo subacuáticas [8].

Además de este sector, los AUVs pueden proveer de información a otros sectores como industrias de telecomunicaciones marinas, instituciones académicas o grupos gubernamentales o privados [26].

Dentro de estos sectores, algunas de las aplicaciones básicas son búsqueda, exploración, mapeo, monitoreo e inspección.

Con el paso del tiempo la tecnología de estas aplicaciones ha avanzado considerablemente, ya que muchos problemas han surgido, algunos ya se han ido resolviendo, otros están desarrollando soluciones y otros problemas surgirán.

Es complejo listar todas las características tecnológicas que conllevan los AUVs, pero algunas de las básicas con las que deben contar son [81]:

- **Autonomía:** Es uno de los puntos más importantes ya que limita de manera notable a los sistemas. El problema de la escasez de autonomía es algo con

lo que se ha lidiado no solo en estos ámbitos, sino en general, sigue siendo el que más investigación conlleva y que por supuesto sigue en constante investigación.

Los problemas más notables son la durabilidad, como también las dimensiones, ya que se requiere mayor aguante de las baterías y menor tamaño, siendo esto, una tarea de los avances tecnológicos, el reducir las dimensiones de éstas para obtener mayores duraciones de energía.

- **Navegación:** Desde los principios los AUV se han operado por medio de una navegación prácticamente ciega. Es por esto que se han desarrollado sistemas acústicos que dan una gran ventaja para evitar posibles golpes o choques contra el fondo o simplemente obstáculos a superar de forma totalmente precisa esta habilidad es llamada *situational awareness*, y también que sean de coste accesible.

  Ahora se han implantado sistemas de navegación inercial combinados con sonar que ofrecen un sistema de navegación mucho más precisos y rápido que los sistemas acústicos, pero a cambio de una gran precisión también se produce un aumento de coste.

- **Sensores:** Los sensores diseñados para los robots terrestres, raramente pueden ser integrados directamente en un AUV ya que no han sido diseñados para su utilización bajo el agua, soportando la presión, temperatura y humedades existentes.

  Por esta razón empresas de diseño de sensores o bien de integrados, han hecho un especial esfuerzo en adaptar los sensores a las especificaciones exigidas por los AUV, como pueden ser la versatilidad, la rapidez, el bajo consumo, la reducción del tamaño o el alto rendimiento.

- **Comunicaciones:** Actualmente podemos encontrar sistemas acústicos que ofrecen una comunicación basada en radiofrecuencia con un error relativo bajo. En los últimos 10 años se ha llegado hasta rangos de Km/s (kilomuestras por segundo) a pocos Kbps, a pesar de que este aspecto de los AUVs sigue en constante mejora e investigación [7].

2.1.3. Antecedentes históricos de los AUVs

Es necesario comprender lo que ha sucedido en las últimas décadas sobre el desarrollo de los AUVs, es por ello que se realizará un breve resumen sobre el desarrollo de esta tecnología a lo largo de los años [3].

El desarrollo de los AUV se inició en la década de 1960. Algunos AUV se construyeron principalmente para aplicaciones muy específicas de recopilación de datos. No obstante, son escasos los artículos publicados acerca de este tipo de submarino [25].

Durante la década de 1970, aparecieron los primeros bancos de pruebas. La Universidad de Washington desarrolló las *APL UARS* [18] y vehículos *SPURV* [13] para recopilar datos de las regiones árticas. La Universidad de la Marina de New Hampshire (en el laboratorio de Ingeniería de Sistemas, actualmente conocido como Sistemas Subacuáticos) desarrolló el vehículo *EAVE* [19] junto con un esfuerzo complementario llevado a cabo en las instalaciones de la Marina de los Estados Unidos, en San Diego.

Figura 3: SPURV [13]

En esta década el número de AUVs era muy reducido. No obstante, este periodo se puede definir como un momento significativo en la experimentación de estos vehículos submarinos.

En la década de 1980 hubo una serie de avances tecnológicos fuera del campo de los AUVs, que en gran medida afectaron al desarrollo de éstos. Los ordenadores pequeños de baja potencia empezaban a ofrecer la posibilidad de implementar algoritmos complejos de guía y control en plataformas autónomas. A pesar de estos avances tecnológicos, se hizo evidente una serie de problemas en el desarrollo tecnológico.

El programa más exitoso en EEUU, se dio en los laboratorios Draper, donde se desarrollaron dos AUVs grandes para ser utilizados como bancos de prueba en una serie de programas de la marina de guerra [20] [21].

Esta década fue de hecho el punto de inflexión para la tecnología de AUVs. Era obvio que la tecnología evolucionaría en los sistemas operativos, aunque no estaba tan claro las tareas que realizarían esos sistemas.

A finales del siglo XX, un número significativo de organizaciones de todo el mundo se comprometió al desarrollo de proyectos basados en AUVs multitarea. En la década de los 90 aparecieron nuevos avances como la utilización del sistema de muestreo Autónomo Oceanográfico (AOSN), que provisionó los recursos necesarios para mover la tecnología a mercados de comercialización.

Hubo un gran avance en cuanto a recursos necesarios para avanzar en la tecnología AUV de cara a la comercialización.

A comienzos del siglo XXI, se puede definir el AUV como producto industrial para obtener un beneficio económico, ya que los mercados están definidos y evaluados en cuanto a su rentabilidad.

Existe una cantidad de compañías que disponen de sistemas AUV comerciales para la venta. Las principales son Kongsberg Hydroid, Bluefin Robotics, Saab y Teledyne [8]. Todas ellas diseñan y construyen submarinos con aplicaciones industriales, investigación marina, comercial o defensa.

Últimamente, se ha mostrado mucho interés en la industria petrolera y de gas, ya que al emplear este tipo de tecnologías, se libran de muchos problemas, costos y riesgos que conllevan las actividades de inspección y muestreo subacuáticas. Esto ha llevado a que las empresas privadas, así como los equipos de las organizaciones mundiales principales, hayan hecho esfuerzos conjuntos para hacer AUVs como parte operativa de la industria del petróleo y del gas, aumentando de esta manera la comercialización y demanda de este tipo de tecnologías.

Un ejemplo de empresas del petróleo que se están uniendo al desarrollo de AUVs es Cepsa. En 2014 comenzó a desarrollar un proyecto con AUVs, el cual se conoce como *Proyecto AUV* y consiste en el desarrollo de un vehículo autónomo submarino para la detección de derrames en las líneas submarinas [9]. La detección temprana de fugas en las líneas submarinas, permite reducir el tiempo de respuesta ante derrames y minimizará por tanto el impacto en el medio marino. Este proyecto está siendo desarrollado por un consorcio liderado por Cepsa, en el cual participa *IXION Industry and Aerospace*, así como la Universidad Complutense de Madrid y se encuentra financiado por el Ministerio de Economía y Competitividad bajo el marco del programa INNPACTO.

También han surgido fundaciones organizadoras de concursos para el desarrollo de robótica submarina, como lo es la AUVSI (*Association for Unmanned Vehicle Systems International*). Cuenta con más de 7500 miembros de organizaciones gu-

bernamentales, industriales y académicas. AUVSI se encarga de fomentar, desarrollar y promocionar la tecnología robótica [4].

Las investigaciones en tecnología del AUV siguen su proceso con el fin de que su desarrollo y comercialización continúen aumentando.

En la actualidad la tecnología en los AUV ha llegado a puntos muy importantes ya que ahora es cuando empiezan a ser más conocidos y están llegando a ser una opción comercial más rentable.

Además, se aprecia una serie de mercados de AUVs emergentes. Aunque no está claramente definido el nivel de interés por parte de personas y organizaciones, hay indicios que sugieren una mayor cota de oportunidades para la comercialización de AUVs en los próximos años.

Las tendencias actuales son las que el mercado ha establecido:

- En primer lugar el desarrollo de dispositivos de bajo coste. Se prevé que estos sistemas, eventualmente, se utilicen en grupos de vehículos cooperantes.

- En segundo lugar los AUV son sistemas más sofisticados que contienen todo tipo de sensores complejos configurados para satisfacer las necesidades específicas del usuario. Aunque no son de bajo coste, pueden acometer tareas, que si se hacen de otro modo, incrementaría el importe de las misiones.

En el ámbito de las ciencias del mar, el potencial de los AUV está claramente reconocido por la mayoría de investigadores. La posibilidad de recopilación de datos marítimos y oceanográficos ha tenido un impacto positivo en la comunidad. De hecho, la preocupación estriba en que se espera demasiado de la evolución de esta tecnología. Los éxitos y fracasos en los próximos años, ayudarán a ajustar la capacidad del sistema y las expectativas de los usuarios. Esto es signo de una tecnología madura. En general se reconoce que la tecnología AUV tiene un papel importante que desempeñar en el futuro los programas de adquisición de datos en el ámbito marítimo.

Prueba de ello es que la Marina de EEUU está fomentando y apoyando un esfuerzo coordinado: el AOSN, proyecto que pretende una conexión en red para adquirir datos oceanográficos por parte de múltiples submarinos autónomos. Tiene como objetivo conseguir una resolución espacial y temporal muy superior a las actuales.

Aunque se haga hincapié en las zonas costeras, a largo plazo se podría prever un sistema similar para obtener información a través de los océanos del mundo. La problemática estriba en la autonomía que sostienen los AUVs.

2.1.4. AUVs comerciales

En esta sección se van a exponer los distintos modelos en base a sus características generales e hidrodinámicas. Para más información, se puede encontrar en el PFC de la referencia [27].

2.1.4.1 Alister AUV

El Alister AUV es un vehículo adaptado para poder abordar misiones a cotas inferiores a 300 m en aguas con diferente salinidad y temperatura. Ha sido desarrollado por ECA.

Su modo operativo varía en función de la plataforma en la que se despliega; para cada una de ellas acometerá una misión distinta. Entre ellas se encuentran el reconocimiento marino (utilizando como plataforma un barco de reconocimiento), misiones oceanográficas e hidrográficas (desplegado de un buque científico) o realizar prevenciones y medidas antiminas a partir de corvetas y fragatas.

El Alister dispone de un sistema de gestión de misiones fácil de utilizar. Un equipo con doble pantalla permite definir las misiones y simultáneamente observar y controlar el estado actual de los sensores instalados en el AUV.

Este vehículo está equipado con dos propulsores longitudinales y timones horizontales y verticales a popa y horizontales a proa. Sus principales características físicas son las que se muestran en la tabla 1.

Eslora	5 metros
Radio	0,7 metros
Peso	980 kg
Velocidad de crucero	4 nudos
Velocidad máxima	8 nudos
Cota	300 metros

Tabla 1: Características del Alister

Figura 4: Alister AUV

2.1.4.2 Hugin 1000

Este dispositivo submarino fabricado por Kongsberg, tiene la peculiaridad de estar disponible para dos profundidades distintas (1000 y 3000 metros). Presenta una estructura modular de tres secciones, siendo la proa y la popa fija, el caso cilíndrico se configura según el tipo de sensor que se requiera para cada misión.

Su diseño está enfocado para ofrecer una baja resistencia hidrodinámica, una elevada estabilidad y amplia maniobrabilidad. El sistema propulsivo muestra un gran rendimiento a bajas velocidades. Así mismo las palas de la hélice, están formados con unos perfiles eficientes y bajos niveles de ruido.

Las características generales de este AUV son las de la tabla 2.

Características	1000 metros	3000 metros
Carena	4,5 metros	4,5 metros
Radio	0,75 metros	0,75 metros
Peso	650 kg	850 kg
Velocidad de crucero	2 nudos	
Velocidad máxima	6 nudos	

Tabla 2: Características del Hugin

Figura 5: Hugin AUV

2.1.4.3 AUV 62-MR

Este vehículo submarino es la última generación de SAAB. Las principales misiones para las que se ha concebido son reconocimiento de minas, mapeado de fondo marítimo, monitorización del entorno y submarino artificial para entrenamiento de objetivo acústico.

El sumergible tiene una fácil adaptación para un fácil lanzamiento desde TLT. No obstante, también se puede lanzar desde plataformas en barcos o desde tierra.

Sus características principales son las que se muestran en la tabla 3.

Eslora	4 - 7 metros
Diámetro	0,53 metros
Peso	650 - 1500 kg
Velocidad de crucero	3 nudos
Velocidad máxima	10 nudos
Cota máxima	500 metros

Tabla 3: Características del AUV 62-MR

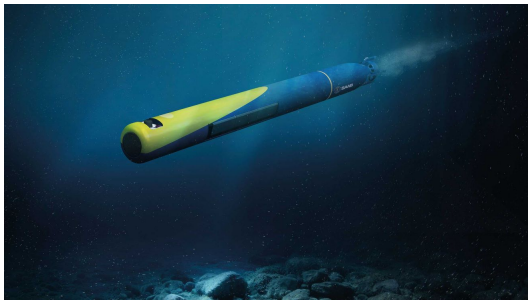


Figura 6: AUV 62-MR

2.1.4.4 REMUS 600

Este sumergible está diseñado para operaciones a grandes profundidades. Presenta una elevada autonomía, permitiéndole un rango amplio de trabajo y unas elevadas prestaciones.

El REMUS 600 viene dotado para poder acometer un gran número de misiones entre ellas estudios hidrográficos, monitorización del entorno, mapeado de restos, operaciones de búsqueda y salvamento y operaciones pesqueras entre otros.

Entre sus características principales se encuentran las indicadas en la tabla 4

Eslora	3,25 metros
Diámetro	0,324 metros
Peso	240 kg
Velocidad de crucero	3 nudos
Velocidad máxima	5 nudos
Cota máxima	600 metros

Tabla 4: Características del REMUS 600

Figura 7: AUV REMUS 600

2.1.4.5 REMUS 100

El REMUS 100 es un sumergible ligero, diseñado para operar en ambientes costeros. Es sin duda, uno de los AUVs más importantes del mercado debido a su gran variedad de misiones. Tiene la peculiaridad, respecto a otros de ser tan pequeño que puede ser transportado por dos personas. No obstante está constituido por una alta gama de sensores sofisticados.

Entre sus principales misiones se puede encontrar: estudios hidrográficos, operaciones de medidas contra minas, operaciones de seguridad portuaria, monitorización del entorno y operaciones pesqueras entre otros.

Las características principales se muestran en la tabla 5.

Eslora	1,6 metros
Diámetro	0,19 metros
Peso	240 kg
Velocidad de crucero	3 nudos
Velocidad máxima	5 nudos
Cota máxima	100 metros

Tabla 5: Características del REMUS 100

Figura 8: AUV REMUS 100

2.1.5. Girona 500

El Girona 500 [47] (conocido por su abreviatura g500) es el robot empleado para resolver el problema planteado. Por consiguiente, es necesario conocer sus características.

Este robot pertenece a la Universidad de Gerona (ViCOROB) y se encuentra en el centro de investigación de la robótica submarina (CIRS, *Research Centre in Underwater Robotics*) también en la Universidad de Gerona. Este recinto está formado por dos edificios; en uno de ellos se encuentran tanto los laboratorios como las oficinas, mientras que en el otro se encuentra el tanque de agua en el que se realizan las pruebas con el submarino y una habitación desde la que observar los movimientos del submarino.

Este robot es un AUV reconfigurable diseñado para funcionar a una profundidad de 500 metros. El vehículo está creado por una estructura de aluminio que soporta 3 cascos con forma de torpedo. Los dos superiores contienen los elementos necesarios para que el submarino flote y los componentes electrónicos, mientras que el inferior contiene los elementos pesados como son las baterías. Esta distribución permite separar el centro de gravedad del centro del centro de empuje separándolos así una distancia de 11 cm. Esta colocación permite al vehículo tener una estabilidad en *pitch* y *roll*, facilitando así las posibles intervenciones que tengan que realizar los operarios desde la plataforma.

Una de las características más remarcables del GIRONA 500 es la capacidad de reconfigurarlo para diferentes misiones. En la configuración estándar está equipado con sensores de navegación como son el DVL (*Doppler Velocity Log*), AHRS (*Attitude and Heading Reference System*) y el USBL (*Ultra-Short BaseLine*) y equipo básico de medición (sonar, cámara y sensor de velocidad). Otra de las características de este vehículo es que cuenta con más de la mitad del espacio de la cápsula inferior del submarino, para añadir distintos sensores en función de la misión. Todos estos sensores serán estudiados en la sección 5.2.3.3.

En la imagen 9 se pueden observar los componentes del submarino.

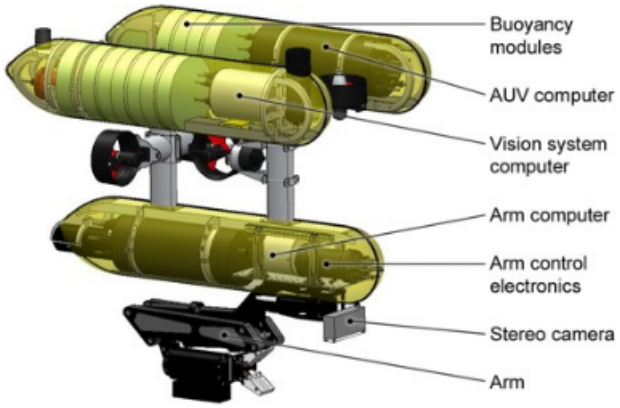


Figura 9: Partes del GIRONA 500

Como se puede observar, existe la posibilidad de incorporar un brazo robótico al submarino, para poder coger objetos que se consideren importantes durante las misiones. A continuación, en la imagen 10 se puede observar el alcance del brazo que se puede añadir al g500.

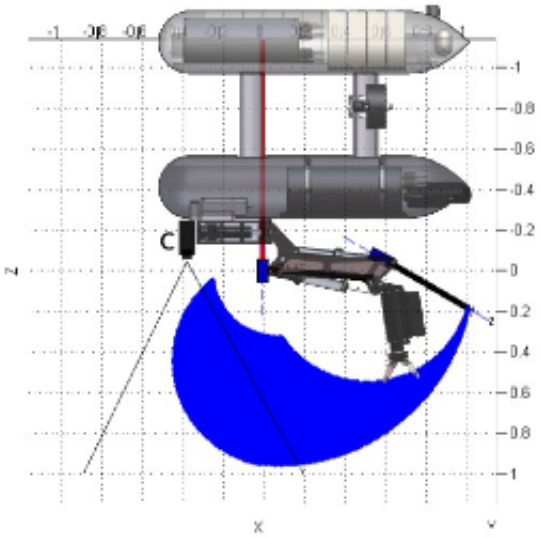


Figura 10: Alcance del brazo del GIRONA 500

Las características de este AUV son las que se indican en la tabla 6.

Eslora	1,5 metros
Diámetro	0,3 metros
Altura	1 metro
Anchura	1 metro
Peso	140 kg
Cota máxima	500 metros
Energía	2.2kWh Li-Ion
Duración de la batería	más de 8 horas

Tabla 6: Características del Girona 500

Este diseño ofrece una buena capacidad hidrodinámica y un gran espacio para almacenar el equipo a la vez que tiene una forma compacta que permite operar el AUV desde barcos pequeños.

El sistema de propulsión cuenta con una configuración básica, tiene cuatro propulsores para controlar los ángulos de navegación, de los cuales dos son verticales para actuar en dirección del *pitch* y dos horizontales para el movimiento de *yaw* y *roll*. No obstante es posible reconfigurar el vehículo para operar solo con 3 propulsores (uno vertical y dos horizontales) y con posibilidad de incrementar el número de propulsores hasta 8 para controlar todos los grados de libertad.

Los ángulos de navegación [11] permiten describir la orientación de un objeto en tres dimensiones. Los ángulos de navegación son un tipo de ángulos de Euler y se utilizan cuando se desea dar la posición del sistema móvil en un momento dado al tener un sistema de coordenadas móvil respecto de uno fijo (en tres dimensiones).

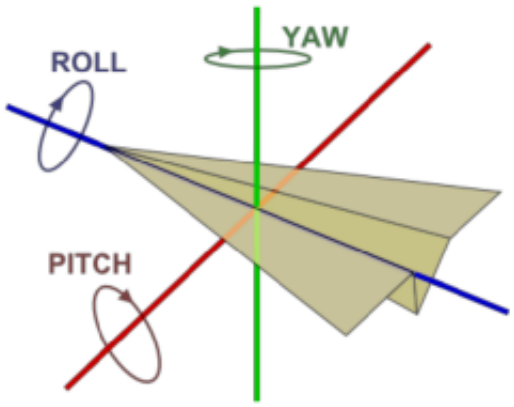


Figura 11: Ángulos de navegación

Como se puede observar en la imagen 11, los tres ángulos son la dirección (*yaw*), elevación (*pitch*) y ángulo de alabeo (*roll*).

2.2. Navegación de un robot

La navegación es la ciencia de conducir un robot móvil mientras atraviesa un entorno para alcanzar un destino o meta sin chocar con ningún obstáculo.

La navegación nos lleva a crear una serie de herramientas totalmente necesarias para la creación de una trayectoria viable y segura. Se ha de crear un mapa con la información que existe o ir creando uno según se explore. Para ello se debe tener en cuenta distintas variables como son la percepción del entorno, fusión de sensores y control de movimiento entre otros.

Planificar es prever y decidir las acciones que permiten alcanzar el objetivo deseable, no se trata de hacer predicciones, sino de tomar las decisiones necesarias para alcanzar ese objetivo.

La idea general en la planificación de trayectorias consiste en desvincular el problema de la cinemática y dinámica del robot del de la obtención de una ruta (óptima o no) libre de obstáculos.

Actualmente existen numerosos algoritmos en tres dimensiones para planificar trayectorias, no obstante los más sencillos de comprender, y que se analizarán en la sección 2.5, son en dos dimensiones, ya que en su mayoría están diseñados para robots que solo pueden maniobrar en el plano cartesiano (ya que no pueden variar la z). Sin embargo, tanto los submarinos como los drones, necesitan incluir una tercera dimensión en la que moverse. No obstante, los algoritmos estudiados son extrapolables a 3 dimensiones, para lo cual habría que resolver tres veces la simulación (una para el plano XY, otro para el plano YZ y finalmente XZ). Analizando las posiciones en las que no hay obstáculos que entrasen en conflicto con otros planos, el robot sería capaz de calcular la trayectoria hasta el objetivo final.

El simulador utilizado para resolver el problema propuesto tiene predefinidos algunos algoritmos que se estudiarán posteriormente en esta memoria (sección 2.5.2).

2.3. Planificación de trayectorias

La planificación de movimiento (*motion planning* o *path planning*) es un termino utilizado en robótica e inteligencia artificial para referirse a la resolución de problemas consistentes en trasladar un objeto desde un punto inicial hasta un punto objetivo sin excederse de unos límites en el espacio y sin colisionar con obstáculos conocidos de antemano.

El origen de la planificación de trayectorias viene de la búsqueda y la necesidad de obtener estrategias de control para que el robot obtenga las trayectorias adecuadas y la mayor calidad en su desplazamiento. Para poder llevar a cabo la

planificación de trayectorias es necesario conocer o ser capaz de obtener el modelo cinemático y dinámico del robot que se pretenda controlar.

Normalmente, la planificación de movimiento no tiene en cuenta las limitaciones físicas del problema como por ejemplo es el peso del robot, sino únicamente su representación en el espacio. Sin embargo existen algoritmos capaces de integrar este tipo de limitaciones no geométricas a la hora de obtener una solución al problema.

Los algoritmos de planificación de movimiento no solo son aplicables a problemas donde un objeto (robot) se mueve de un punto a otro en el espacio. Este campo de la inteligencia artificial ha demostrado ser muy útil también en otras disciplinas.

A pesar de que la mayoría de aplicaciones de estos métodos se dan en robótica [37] [38], también son muy utilizadas en campos tan aparentemente alejados como la biología computacional, donde se usan para el estudio del comportamiento de las proteínas [39]. Dentro de la robótica, estos algoritmos no solo sirven para desplazar un robot de un lugar a otro en el espacio, sino que son capaces de planificar cualquier movimiento desde la fabricación en línea con robots industriales hasta la resolución de un cubo de Rubik.

En sus inicios, los algoritmos de planificación de movimiento tenían como objetivo asegurar encontrar una solución si ésta existía. Los métodos actuales han resuelto este problema satisfactoriamente y ahora la investigación en este campo se centra en la optimización de la obtención de soluciones (en tiempo, distancia, tiempo de computo, etc) dependiendo de los requerimientos de cada caso particular.

Actualmente, la planificación de trayectorias continúa siendo un campo activo en la investigación, ya que siguen apareciendo nuevas aproximaciones al problema que intentan mejorar a las anteriores. No se ha encontrado un método que sea mejor que el resto en todos los casos. Aquellos que no requieren una alta carga computacional suelen ofrecer resultados significativamente inferiores a los métodos más complejos, proporcionando trayectorias poco suavizadas o más largas. Mientras que, por lo general los métodos con un mayor coste computacional son mejores en problemas con pocas dimensiones.

Cuando se tienen espacios más complejos el tiempo de cómputo se eleva de forma exponencial, haciendo que los algoritmos que ofrecen mejores resultados en dos y tres dimensiones sean prácticamente inutilizables en aplicaciones como el movimiento de un brazo robótico con varias articulaciones.

La planificación de trayectorias tiene distintos enfoques, y en función del objetivo final se utilizarán unos algoritmos u otros. Los cuatro enfoques más diferen-

ciados son navegación (*navigation*), alcance (*coverage*), localización (*localization*) y mapeo (*mapping*).

- La **navegación** es el problema de encontrar un movimiento sin colisiones del punto de partida al objetivo final del robot (la principal aplicación sería un brazo robótico).
- El **alcance** es el problema de pasar un sensor por todos los puntos del espacio (un ejemplo de ello sería para pintar).
- La **localización** consiste en que el robot utilice un mapa para determinar su posición.
- El **mapeo** es el problema de explorar un entorno desconocido.

En la siguiente tabla se muestran los cuatro enfoques y el tipo de algoritmos que se utiliza para cada situación.

Función	**Robot**	**Algoritmo**
Navegación	Configuración del espacio y grados de libertad	Movimientos óptimos o no óptimos
Mapeo	Cinemática/ dinámica	Complejidad computacional
Alcance	Omnidireccional o restricciones de movimiento	Completud (resolución)
Localización		En línea/ fuera de línea Basado en sensores

Tabla 7: Enfoques de la planificación de trayectorias

2.4. Métodos de planificación de movimiento

Existen varias aproximaciones distintas a la planificación de trayectorias. Actualmente no se ha llegado a un convenio en la clasificación de estos métodos. Esto es debido a la gran variedad de contextos en los que pueden ser aplicados, lo que hace que resulte complicado establecer fronteras entre ellos.

La Valle propone en [40] una clasificación de estos algoritmos en dos grupos en función de la forma en que manejen la información: métodos combinatorios (*combinational planning*) y métodos basados en muestreo (*sampling-based planning*):

- **Métodos combinatorios:** Son aquellos que construyen estructuras de datos que representan todo el *state space* de forma discreta. Pueden resolver cualquier problema de planificación de movimiento. Esto significa que son métodos completos, es decir, la obtención de una solución está garantizada

si esta existe además de la correcta identificación de problemas donde no exista solución.

A cambio de esta eficacia, los métodos combinatorios tienen una mayor complejidad computacional, y la representación del espacio completo hace que estos algoritmos sean ineficientes a la hora de tratar con un número elevado de dimensiones. Un ejemplo de este grupo serían aquellos algoritmos como *Dijkstra* [41] o $A*$ [42], que discretizan el espacio en una rejilla de n dimensiones.

- **Métodos basados en muestreo aleatorio:** Se refiere a aquellos algoritmos que aplican métodos de detección de colisiones para muestrear una serie de estados aleatorios dentro del *free space*. Una vez obtenido esto, se busca un camino hasta la solución uniendo varios de estos estados. En este caso, no se garantiza la completa resolución del problema, sino que solo es posible asegurar la probabilidad con la que se obtendrá la solución. Además, las muestras aleatorias no siempre obtienen el camino más óptimo.

  Estos métodos tienen un coste computacional mucho menor que los algoritmos combinatorios, sobre todo para espacios grandes con alto número de dimensiones. Su eficacia depende en gran medida del número de estados muestreados, por lo que la eficacia se podría mejorar permitiendo al algoritmo más tiempo para obtener estados aleatorios.

Además de estos dos grandes grupos propuestos por La Valle, se podría ampliar esta clasificación con otro tipo de algoritmos que se pueden considerar suficientemente diferentes del resto como para no pertenecer a ninguno de estos grupos: los métodos basados en control (*control-based methods*).

- **Métodos basados en control:** Estos métodos se basan en la aplicación de las ideas de la teoría de control para guiar al robot a su objetivo. Normalmente tienen un bucle realimentado que se aplica continuamente hasta obtener la solución.

  Estos métodos tienen la ventaja de ser poco costosos computacionalmente, sin embargo no suelen obtener las mejores soluciones, especialmente en entornos complejos. Una gran ventaja de estos métodos es que gracias a su velocidad pueden ser aplicados de manera online, lo cual es necesario en algunos casos.

No existe un tipo de algoritmo que sea superior al resto en todos los aspectos, sino que cada uno tiene sus fortalezas y debilidades. Dependiendo del problema a resolver y de las características y limitaciones propias de cada caso concreto se obtendrán mejores resultados aplicando uno u otro método.

2.5. Algoritmos de planificación de trayectorias

En la actualidad se pretende dotar a los robots de una mayor autonomía, no solo a nivel de la batería, sino de la capacidad de que puedan operar durante un largo periodo de tiempo sin la necesidad de un operario, consiguiendo así que el usuario únicamente tenga que introducir el objetivo inicial y final y sea el propio robot el que calcule la trayectoria para llegar hasta allí.

Una vez que el objetivo y el robot han sido explicados, se puede analizar entre los distintos algoritmos cual es el más óptimo en función del método empleado para resolver el problema y el tiempo que emplean en encontrar la solución.

Algunos algoritmos siempre encuentran la solución en caso de que exista; si no existe una solución se notificaría al usuario. Esta propiedad es muy importante, ya que al incrementar el número de grados de libertad, encontrar algoritmos capaces de alcanzar siempre el objetivo puede ser de gran complejidad.

Existen algoritmos *online* y *offline*. Los algoritmos *offline* son aquellos que capaces de construir el plan por adelantado ya que se basa en conocer el entorno con anterioridad. El otro tipo de algoritmo es aquel que va construyendo el recorrido conforme se va moviendo y encontrando obstáculos a su paso.

En este caso, para la parte práctica del proyecto, se conoce el entorno donde se moverá el submarino; por lo que en la resolución del problema se emplearán algoritmos *offline*.

Existe una gran variedad de algoritmos de planificación de movimiento, como se puede ver en la tabla 8.

Determinísticos	Basados en grafos	
	Campos potenciales artificiales	
Probabilísticos y aleatorios	Planeador aleatorio de trayectorias	
	Mapas probabilísticos	PRM
	Arboles de exploración rápida	RRT
	Basados en optimización	Algoritmos genéticos (GA)
		Colonia de hormigas (ACO)
		Enjambre de partículas (PSO)
		Quimiotaxis bacteriana (BC)
Basados en trayectoria	Bug	Bug1
		Bug2
		Tangent Bug
	Grafos de visibilidad	Dijkstra
Basados en celdas	Descomposición en celdas exactas	
	Descomposición en celdas aproximadas	
Diagrama de Voronoi		
Modelado de espacio libre		

Tabla 8: Tipos de algoritmos

Algunos de los algoritmos mencionados anteriormente, se explicarán en la sección 2.5.2 entre otros muchos comprendidos en la librería OMPL.

En primer lugar se realizará un estudio de los algoritmos OMPL (*Open Motion Planning Library*), ya que son los que se emplean en la librería de *MoveIt* y a continuación se describirán los de la tabla 8.

2.5.1. Conceptos

En esta sección se va a presentar la terminología básica del control de movimiento que van a ser utilizados a lo largo del trabajo. Dado que la inmensa mayoría de la investigación, así como las bibliotecas existentes se encuentran en inglés, los conceptos no han sido traducidos al Castellano.

- ***Workspace:*** El espacio físico donde opera el robot. Los límites de este espacio no pueden ser atravesados, por lo que se representan como obstáculos.

- ***State:*** Es una posición concreta del robot en el espacio. Esta representada por un calor en cada uno de los grados de libertad. También se puede llamar configuración.

- ***State space:*** Es el conjunto de todas las configuraciones posibles del robot dentro del *workspace*.

- ***Free state space:*** Es un subconjunto del *State space* que contiene todos los estados en los que el robot no está colisionando con ningún obstáculo.

- ***Path:*** En español camino o trayectoria. es un conjunto de configuraciones del robot dentro del *state space*. Un camino es válido si cada uno de sus estados se encuentran dentro del *free state space*.

Partiendo de estas definiciones, se puede reformular el problema como la tarea de encontrar un camino (*path*) válido dentro del *free state space* del robot entre un estado inicial y uno final dados.

Otros conceptos necesarios para asegurar la correcta comprensión de los algoritmos son los que se indican a continuación.

- **Configuración de un robot:** es el conjunto mínimo de parámetros capaces de identificar de forma única la posición y orientación de todos y cada uno de los sólidos rígidos que componen el robot.

- **Configuración origen:** es la configuración que presenta el robot antes de iniciar cualquier acción. En el problema de planificación es el punto de partida.

- **Configuración destino:** es la configuración que se desea alcanzar partiendo de la anterior.

- **Espacio de configuraciones:** es el conjunto de todas las configuraciones que puede adoptar el robot independientemente del escenario en el que se ubique. La dimensión de este espacio será igual al número de parámetros necesarios para describir una configuración. Habitualmente se denota como C.

- **Colisión:** se dice que un robot entra en colisión cuando se configuración sitúa uno o varios de sus elementos intersecando el espacio ocupado por obstáculos o los límites del escenario de estudio.

- **Obstáculo:** para el ámbito de la planificación, un obstáculo se considera como un conjunto cerrado y acotado de puntos del espacio físico. Es decir, sólo interesa del obstáculo el volumen que ocupa y no su naturaleza. Así, un obstáculo también puede ser un lugar que por alguna razón no se desea que el robot ocupe.

- **Obstáculo C:** sea q un elemento de C, denotamos como $A(q)$ el espacio físico que ocupa el robot cuando adopta la configuración q. Sea B_i un obstáculo, se define el obstáculo C asociado a B_i (abreviadamente CB_i), como el conjunto de configuraciones cuyo espacio físico colisiona con B_i. La existencia de colisión del robot con un obstáculo implica que el espacio que ocupa $A(q)$ es compartido con el que ocupa el obstáculo B_i.

- **Región de obstáculos C:** es la unión de todos los obstáculos C y se denota como CB.

- **Espacio de configuraciones libres de colisión:** es el subconjunto del espacio de configuraciones cuyos elementos no presentan colisión. Habitualmente se denota como C_{free}.

Muchos de los métodos que se describen a continuación, utilizan el modelo de robot puntual (constituido por un único cuerpo de tamaño nulo) cuya configuración consiste en la posición cartesiana del mismo.

Aunque los diagramas y modelos en su mayoría utilizan este modelo, esto no quiere decir que las capacidades de estos métodos se limiten al mismo. Muy al contrario la mayoría pueden extrapolar sus capacidades a espacios de configuración de dimensión elevada.

2.5.2. Librerías y algoritmos de *MoveIt*

Existe la posibilidad de emplear diferentes familias de algoritmos en función del problema particular que se quiere resolver.

MoveIt está diseñado para trabajar con distintos planificadores de trayectorias, algunos de ellos se muestran en la lista a continuación, en orden descendente en función de la popularidad de estos [74] .

- **OMPL (Open Motion Planning Library):** OMPL es una librería *open source* que principalmente implementa aleatoriamente algoritmos para obtener una trayectoria de la posición inicial a la final.

- **STOMP (*Stochastic Trajectory Optimization for Motion Planning*):** STOMP es un planificador de trayectorias óptimo basado en el algoritmo PI^2 (Policy Improvement with Path Integrals). Puede planificar trayectorias evitando obstáculos y optimizando las limitaciones.

- **SBPL (*Searched-Based Planning Library*):** es un planificador de trayectorias basado en la discretización del espacio.

- **CHOMP (*Covariant Hamiltonian Optimization for Motion Planning*):** esta librería se basa en la optimización de trayectorias mediante un gradiente. Este algoritmo capitaliza un gradiente covariante y uno funcional para aproximarse al estado de optimización.

MoveIt (sección 5.3), es flexible y modular, por lo que puede trabajar con cualquier algoritmo de planificación de trayectorias con la apropiada interfaz.

La librería OMPL [36] [49] se compone de muchos algoritmos de planificación de muestreo basado en le movimiento. OMPL en sí no tiene ningún código relacionado con el control de colisión o de visualización. Esta es una opción de diseño deliberada, de modo que OMPL no está vinculada a un corrector de colisión frontal en particular o la visualización, solo entrega una trayectoria a seguir.

Además de la librería OMPL, hay otra que cumple con la interfaz descrita (SBPL [48]). Ambas librerías son independientes de *ROS* y agnósticas al hardware que modelan. *SBPL* usa algoritmos basados en primitivas y OMPL basados en muestras.

Dado que la librería utilizada por defecto por *MoveIt* (y por tanto el empleado en la resolución del problema) es OMPL; a continuación se profundiza en los distintos algoritmos que utiliza.

2.5.2.1 Algoritmos basados en primitivas

Consisten en movimientos básicos que se encadenan para formar movimientos más complejos. Se realiza una búsqueda en cada estado entre un subconjunto de movimientos posibles para el robot considerando sus restricciones y los obstáculos, lo que permite construir el grafo incrementalmente o pre-calcularlo dependiendo de las necesidades del problema. La librería SBPL implementa varios de estos algoritmo, siendo los más relevantes los que se presentan en las siguientes secciones.

Este tipo de algoritmos presenta ciertas ventajas y desventajas:

- Ventajas:
  - El grafo generado es pequeño.
  - Todos los caminos encontrados son factibles.
  - Incorpora naturalmente las restricciones del robot u otras adicionales que sean de interés.

- Desventajas:
  - Dependiendo de la discretización elegida puede que sea imposible muestrear ciertos caminos válidos. Esto hace concluir falsamente infactibilidad para un problema factible.

2.5.2.1.1. Anytime Repairing A* (ARA*)

A* [50] expande sus estados basándose en la minimización $g + h$, donde g es el costo conocido de llegar a un nodo y h es una heurística de cuánto cuesta llegar a la meta desde dicho nodo. A* ponderado minimiza $g + \epsilon h$ con $\epsilon > 1$, garantizando que:

$$costo\,(solucion) \leq \epsilon \cdot costo\,(solucion\,optima)$$

La idea de ARA* es correr A* varias veces disminuyendo ϵ hasta que se acabe el tiempo y re-usar los valores de los estados que no cambian. La figura 12 muestra una ejecución de ARA* comparándolo con A* ponderado, donde se ve que A* ponderado debe repetir la búsqueda en cada caso pero ARA* solo expande los estados no visitados aumentando considerablemente la eficiencia.

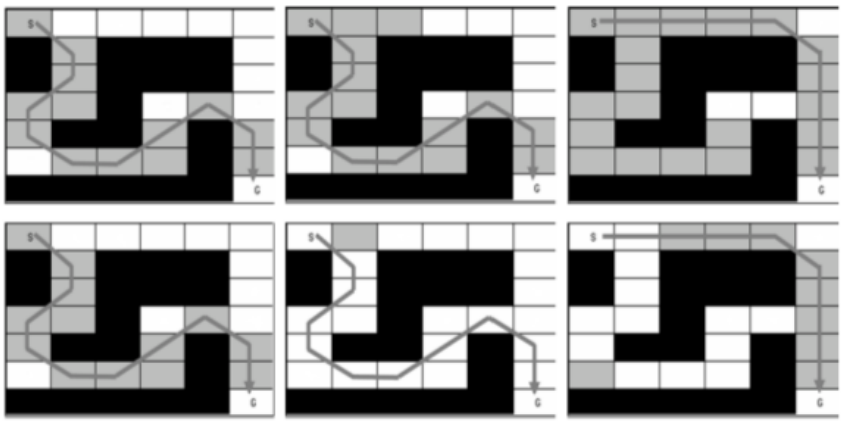


Figura 12: Ejecución del algoritmo para encontrar una trayectoria de S a C. De izquierda a derecha: valores decrecientes de ϵ. Los cuadros negros son obstáculos, los blancos es camino inexplorado los grises representan el camino explorado. Arriba: serie de A* ponderado, abajo ARA*

2.5.2.1.2. Anytime D*

D* [51] es similar ARA* pero funciona sobre entornos dinámicos. Permite re-planificar eficientemente frente a cambios en el camino. Cuando vuelve a planificar puede incrementar o disminuir ϵ.

2.5.2.1.3. R*

R* [52] es la versión aleatorizada de A*. Muestrea caminos intermedios para llegar a la meta. Si un camino está tomando demasiado tiempo, lo evita y sigue con los más sencillos. Sirve para trabajar en entornos complejos y grandes.

2.5.2.2 Algoritmos basados en muestras

Los algoritmos que utiliza la librería OMPL son algoritmos que eligen puntos aleatorios del espacio y los van agregando a la trayectoria actual si están libres de obstáculos. Estos algoritmos se basan principalmente en *Probabilistic Roadmaps* [53] y *Rapid Exploring Random Trees*, sin embargo hay muchos más que no se relacionan directamente con ellos. A continuación se presentan brevemente los más importantes.

2.5.2.2.4. Probabilistic RoadMaps

En este tipo de algoritmos (PRM [53]) se construye un camino con puntos de control válidos (espacio sin obstáculos) que aproxima la conectividad del espacio de estados. Los puntos de control cercanos se conectan con una discretización de movimientos válidos intermedios. Finalmente se usa un algoritmo como A* para buscar un camino en la red creada.

Existen variantes de este algoritmo:

- **PRM* [54]:** En vez de elegir a mano un número de vecinos más cercanos a unir en la red, se eligen el número automáticamente basándose en parámetros de ocupación del espacio de estados, esto quiere decir que el número de vecinos es una función de la razón de espacio libre (sin obstáculos) al espacio total. Esto garantiza soluciones óptimas.
- **Lazy PRM/ Lazy PRM* [55]:** Se comprueban perezosamente las colisiones con el fin de obtener soluciones probables más rápidamente. Esto quiere decir que se encuentra un camino, inicialmente línea recta, y luego este se descarta si presenta colisiones, y se tratan de encontrar un camino que una los segmentos separados por las colisiones.

2.5.2.2.5. SPARSE y SPARSE2

Este tipo de algoritmo [56] [57] construye iterativamente la red de PRM, pero decrece en el tiempo la probabilidad de agregar nodos a la red, lo que garantiza soluciones cercanas a la óptima. La segunda versión obtiene mejores soluciones pero es más lenta que la primera.

2.5.2.2.6. Rapidly-exploring Random Trees (RRT)

La idea de este algoritmo [58] es construir incrementalmente un árbol que determine los caminos posibles. Para agregar un nodo nuevo se muestra un estado al azar q_r y se busca el estado q_n en el árbol que está más cercano al estado muestreado. Luego se expande el árbol un delta ϵ, creando un nuevo estado q_{nuevo} que está más cerca de q_r, hasta que se alcance un estado q_{final}. Finalmente q_{final} se agrega al árbol de exploración. La figura 13 ejemplifica este proceso. Como es un

algoritmo aleatorizado se pueden obtener diferentes soluciones como se ve en la figura 14.

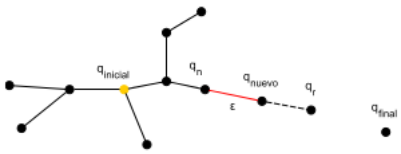


Figura 13: Diagrama ejecución RRT

Figura 14: RRT encontrando camino entre 2 puntos. Caminos encontrados para 2 ejecuciones diferentes de RRT; son diferentes debido a la aleatoriedad del algoritmo

Variantes:

- **RRT Connect [59] y Parallel RRT (pRRT):** Usa dos árboles en paralelo, uno en el inicio y otro en el final e intenta unirlos. La versión paralela es una implementación multiproceso.
- **Lazy RRT [60]:** Al acercarse al nuevo estado q_m, no se comprueban las colisiones. Cuando se encuentra un camino se verifica su validez y remueven segmentos inválidos, volviendo a la búsqueda de antes.
- **RRT* [61]:** Garantiza converger a una solución óptima y el tiempo de cálculo (t) está acotado por $\alpha \cdot t$ (RRT).
- **Lower Bound Tree RRT (LBTRRT) [62]:** Mantiene un grafo y un árbol sobre el espacio de estados. La solución cumple:

$$solucion\,(LBTRRT) \leq \alpha \cdot optimo\,(RRT)$$

- **TRansition-based RRT (T-RRT) [63]:** Considera un mapa de costos que permite considerar obstáculos difusos.

2.5.2.2.7. Expansive Space Trees (EST)

En este algoritmo [64] se comienza con 2 árboles: uno en el estado inicial y el otro en el estado final y se define una rejilla sobre una proyección del espacio de estados (disminución de dimensionalidad). Con esto intenta detectar las áreas menos exploradas siguiendo 2 etapas:

- Expansión
  - $w(x)$ = número de nodos en el árbol que yacen en la vecindad del nodo x.
  - Elegir un nodo con probabilidad $\frac{1}{w(x)}$.
  - Muestrear la vecindad del nodo y agregar un nodo si el camino está conectado a la raíz.
- Conexión
  - Para cada nodo en el árbol inicial, buscar un nodo en el árbol del objetivo.
  - Si la distancia entre los nodos es menor que cierta constante, unirlos y terminar.

2.5.2.2.8. Single-query Bi-directional Lazy collision checking planner (SBL) y su versión paralela (pSBL)

Ambos algoritmos [65] tienen dos árboles para la exploración, similar a un EST con una distancia decreciente de vecindad. La exploración se guía con una rejilla de estados previamente visitados, donde las casillas menos llenas tienen más probabilidades de ser seleccionadas para exploración. Conexión de estados de diferentes árboles es intentada si caen en la misma celda de la rejilla.

2.5.2.2.9. Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE)

Este algoritmo [66] se basa en árboles. Realiza una discretización de varios niveles de diferente resolución para guiar la exploración del espacio continuo, iniciando dinámicamente la rejilla en una proyección del espacio. Se intentan explorar las celdas de la frontera, es decir, con menos de 2 vecinos (se usa vecindad 4: los vecinos son los de los lados y arriba y abajo), y para esto se van eligiendo las celdas de niveles superiores, y se va bajando hasta llegar a las de nivel más bajo. Funciona mejor que RRT, EST y Path-Directed Subdivision Trees (PDST) en varios casos.

Las variaciones más importantes son Bi-directional KPIECE (BKPIECE) que utiliza dos árboles de exploración en los estados iniciales y finales, y Lazy Bi-directional KPIECE (LBKPIECE) que agrega un chequeo de colisiones perezoso. Este último es el usado por defecto en MoveIt.

2.5.2.2.10. Search Tree with Resolution Independent Density Estimation (STRIDE)

Este algoritmo [67] utiliza Geometric Near-neighbor Access Tree una estructura de datos recursiva para optimizar la búsqueda y detectar las áreas menos exploradas. Su funcionamiento es similar a EST pero no requiere una proyección del espacio ni una rejilla dado que la estructura se adapta al espacio de estados.

Está diseñado para trabajar en entornos de alta dimensionalidad, donde supera ampliamente a los otros métodos.

2.5.3. Algoritmos de tipo *Bug*

Los algoritmos de tipo Bug [90] son los más conocidos para resolver el problema de la navegación en un entorno de dos dimensiones desconocido. Se implementan de manera directa y garantizan alcanzar el objetivo establecido (en caso de que no fuese posible alcanzarlo se detendrían). Este tipo de algoritmos requieren dos tipos de comportamiento: moverse en línea recta y cuando el robot se encuentre un obstáculo en el camino, seguir el contorno de éste hasta continuar otra vez en línea recta hacia el objetivo.

Los dos primeros algoritmos asumen sensores táctiles. Sin embargo, *Tangent Bug* cuenta con un detector de distancia finita.

Los algoritmos de tipo *Bug* son más rápidos en cuanto a alcanzar la meta.

2.5.3.1 *Bug1*

Este algoritmo formaliza la idea de moverse en dirección al objetivo y rodear el obstáculo. El robot en este tipo de algoritmos se asume que es un punto con posicionamiento perfecto (sin error) con un sensor de contacto que permite detectar el contorno del obstáculo al entrar en contacto con éste. El robot además puede medir la distancia entre dos puntos x e y $d(x, y)$, asumiendo que el espacio está acotado.

El hecho de que el espacio esté limitado implica que para cualquier x perteneciente al contorno del obstáculo ($x \epsilon C$,) existe un $r < \infty$ que $C \subset B_R(X)$.

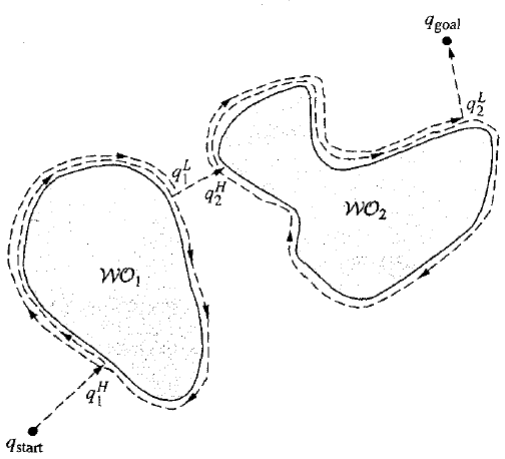


Figura 15: Algoritmo: Bug1

Como se puede observar en la imagen 15, siguiendo el algoritmo Bug1, el robot se mueve siguiendo la línea recta en dirección al objetivo (ya que es la trayectoria más corta) hasta que se encuentra un obstáculo (q_i^H, la H es de *hit*) o llega al punto final (q_{goal}). En el caso de encontrarse un obstáculo q_i^H (punto donde el robot colisiona con el obstáculo y por ello se denomina punto de colisión). A continuación el robot rodea el obstáculo entero (en sentido de las agujas del reloj) hasta que vuelve al punto de partida (q_i^H), donde determina el punto más cercano al objetivo en el perímetro del obstáculo y se dirige a este punto (q_i^L, la L es *leave*). Desde este punto el robot se vuelve a dirigir al objetivo en linea recta hasta llegar al objetivo o en caso de encontrarse un obstáculo se repetiría este proceso tantas veces como fuese necesario para alcanzar la meta. En caso de que el objetivo fuese inalcanzable se notificaría al usuario.

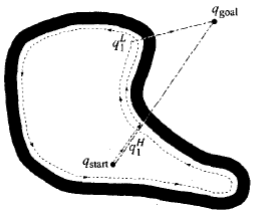


Figura 16: Algoritmo Bug1 con un objetivo inalcanzable

Un ejemplo de situación donde el objetivo es inalcanzable es el mostrado en la figura 16 donde el robot recorre todo el contorno sin encontrar una "salida"para alcanzar la meta.

A continuación se explica el pseudocódigo del algoritmo.

Algorithm 1 Bug 1 Algorithm

Input: Robot equivalente a un punto y un sensor táctil.
Output: Trayectoria hasta el objetivo (q_{goal}) o la conclusión de que no se puede alcanzar éste.

while Forever **do**
  repeat
    Desde q_{i-1}^L, moverse hacia q_{goal}.
  until q_{goal} es alcanzado **or**
    se encuentra un obstáculo en q_i^H.
  if Se alcanza el objetivo **then**
    Detenerse.
  repeat
    Sigue el contorno del obstáculo
  until Se alcanza q_{goal} **or**
    se re-encuentra con q_i^H
    Determinar el punto q_i^L del perímetro que tiene la menor distancia al objetivo.
  Ir a q_i^L
  if el robot se mueve hacia el objetivo **then**
    Concluir que q_{goal} no es alcanzable y detenerse.
end while

2.5.3.2 *Bug2*

El algoritmo Bug2 [90], al igual que en Bug1, hay dos comportamientos: moverse en dirección a la meta y bordear el obstáculo. La principal diferencia entre estos dos algoritmos es que Bug1 hace una búsqueda exhaustiva del punto óptimo del que separarse del obstáculo mientras que Bug2 utiliza un acercamiento oportunista. El objetivo del robot de este algoritmo, al igual que en Bug1, es continuar en la linea recta (*m-line*) que une el punto de partida q_{start} con el punto final q_{goal}.

Otra diferencia entre este algoritmo y el explicado en el apartado anterior es que en este caso el robot no rodea el obstáculo entero, sino que lo hace hasta que vuelve a encontrarse en la línea recta (conocida como *m-line*) más cercana al objetivo. Este proceso se repite hasta que el robot alcanza la meta (q_{goal}), como se puede ver en la imagen 17; o hasta que vuelve al punto de partida (q_{start}, imagen 18); en cuyo caso el robot se detendría y notificaría al usuario que el objetivo es inalcanzable.

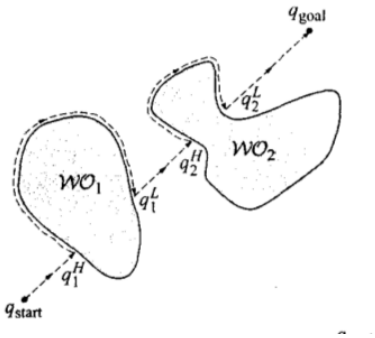


Figura 17: Algoritmo Bug2

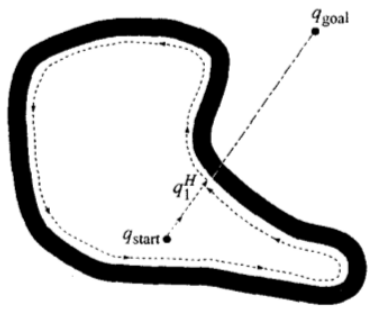


Figura 18: Algoritmo Bug2 con un objetivo inalcanzable

Siendo $x \epsilon C_{free} \subset R^2$, la posición actual del robot $i = 1$ y q_0^L la posición inicial, el algoritmo es el mostrado a continuación.

Algorithm 2 Bug 2 Algorithm

Input: Robot equivalente a un punto y un sensor táctil.
Output: Trayectoria hasta el objetivo (q_{goal}) o la conclusión de que no se puede alcanzar éste.

while True **do**
  repeat
    Desde q_{i-1}^L, moverse hacia q_{goal} sobre la *m-line*.
  until q_{goal} es alcanzado **or**
    se choca con un obstáculo en q_i^H.
    Girar a la izquierda (o a la derecha).
  repeat
    Seguir el contorno
  until q_{goal} es alcanzado **or**
    se choca con un obstáculo en q_i^H **or**
    el robot se reencuentra con la *m-line* en el punto m, el cual $m \neq q_i^H$
    (el robot no alcanza el objetivo), $d(m, q_{goal}) \prec d(m, q_i^H)$, y
  if el robot se mueve hacia la meta, **then**
    no debería chocarse con ningún obstáculo.
  Let $q_{i+1}^L = m$
  Incrementar i
end while

A simple vista este algoritmo parece más eficiente que Bug1, ya que el robot no tiene que rodear todo el obstáculo; no obstante esto no siempre es cierto, ya que depende de la complejidad del obstáculo encontrado en la trayectoria.

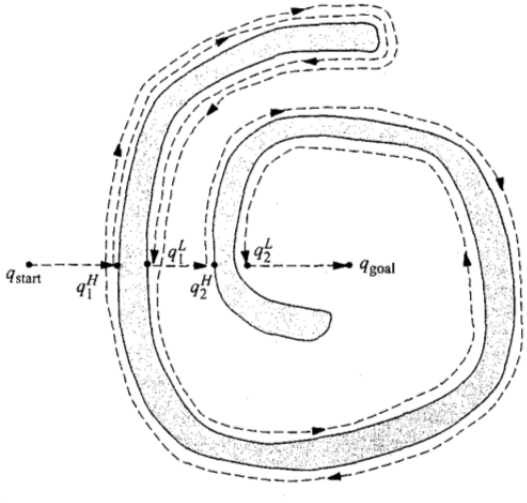


Figura 19: Bug1 vs Bug2

2.5.3.3 Tangent Bug

Este algoritmo es una mejora del Bug2, ya que determina el camino más corto a la meta utilizando un sensor de rango con una resolución de 360° infinita. En ocasiones esta orientación se conoce como *azimut*. Este sensor de rango se modela con una distancia bruta $\rho : \mathbb{R}\ x\ S^1 \rightarrow \mathbb{R}$. Considerando el robot situado en la posición $x\ \epsilon\ \mathbb{R}^2$ con rayos salientes de él. Para cada $\theta\ \epsilon\ S^1$, el valor $\rho(x, \theta)$ es la distancia al objeto más cercano en la dirección del rayo desde el punto x con un ángulo θ. Más formalmente:

$$\rho(x, \theta) = \min_{\lambda \epsilon (0,\infty)} d(x, x + \lambda[\cos\theta, \sin\theta]^T) \tag{1}$$

en el que:

$$x + \lambda[\cos\theta, \sin\theta]^T \epsilon \cup_i WO_i$$

Existen infinitas soluciones a este problema ya que hay muchas $\theta\ \epsilon\ S^1$. Esta expresión es correcta de aproximar con un numero finito de sensores de rango situados alrededor del robot circular móvil (que ha sido modelado como un punto).

Como los sensores de rango tienen un rango limitado, se define como la distancia bruta (ρ: $\mathbb{R}^2$ x $S^1 \Rightarrow \mathbb{R}$, el cual tiene el mismo valor de ρ cuando el obstáculo está dentro del área de sensibilidad; y tiene un valor infinito cuando las longitudes de los rayos son mayores que el radio del sensor de rango (R); lo que quiere decir que los objetos están fuera del rango en el que el robot es capaz de detectar.

$$\rho_R(x,\theta = \begin{cases} \rho(x,\theta), & \text{si } \rho(x,\theta) < R \\ \infty, & \text{en el resto de situaciones.} \end{cases}$$

El algoritmo de Tangent Bug asume que el robot puede detectar discontinuidades en ρ_R. Para un $x \epsilon \mathbb{R}^2$ fijos, se define un intervalo de continuidad, el cual conecta el conjunto de puntos que se encuentran en el contorno del espacio libre $(x + \rho(x,\theta)[\cos\theta, \sin\theta]^T)$ donde $\rho_R(x,\theta)$ es finito y continuo con respecto a θ.

Los puntos finales de estos intervalos tienen lugar cuando $\rho_R(x,\theta)$ pierde la continuidad debido a un objeto bloqueando la trayectoria o un sensor alcanzando su límite de rango.

Al igual que en el caso de *Bug1* (sección 2.5.3.1) *Bug2* (sección 2.5.3.2); *Tangent Bug* tiene dos comportamientos, moverse hacia el objetivo y seguir el contorno del obstáculo. La principal diferencia reside en este segundo comportamiento.

Inicialmente el robot se mueve en dirección al objetivo en linea recta hasta que el sensor de rango detecta un objeto a una distancia R (que se corresponde con el radio de la circunferencia que es capaz de abarcar el sensor) entre él y la meta. Cuando el robot detecta un objeto intenta que el circulo de radio del sensor de radio R sea tangente al objeto; e inmediatamente después este punto tangente se separa en dos $O_i's$, que son los puntos finales del intervalo. En caso de que el obstáculo estuviese en frente del robot, este intervalo interseccionaría con el segmento que conecta el robot y el objetivo. Una vez determinado el segmento O_i, el robot se moverá hacia el O_i que decrezca la distancia heurística a la meta.

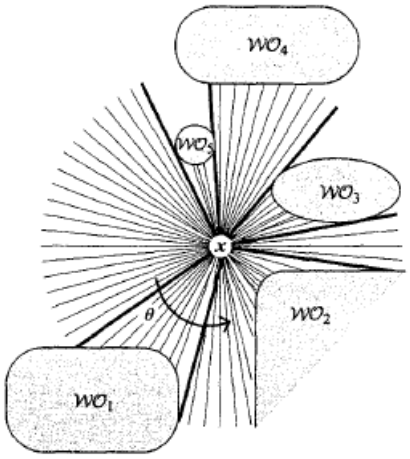


Figura 20: Tangent Bug

Cuando $d_{reach} < d_{followed}$, el robot termina de bordear el obstáculo. Siendo T el punto donde el círculo centrado en x de radio R que conecta x y q_{goal}, T sería el

punto de la periferia del sensor de rango más cercano al objetivo cuando el robot está situado en x. Siendo $x = q_{start}$ y $d_{leave} = d(q_{start}, q_{goal})$ el pseudocódigo se muestra a continuación.

Algorithm 3 Tangent Bug Algorithm

Input: Robot equivalente a un punto y un sensor de rango.
Output: Trayectoria hasta el objetivo (q_{goal}) o la conclusión de que no se puede alcanzar éste.

while True **do**
  repeat
    Moverse de manera continuada al punto $n \epsilon (T, O_i)$, la cual minimiza distancia $d(x, n) + d(n, q_{path})$.
  until
    • q_{goal} es alcanzado **or**
    • la dirección que minimiza la distancia $d(x, n) + d(n, q_{path})$ empieza incrementar $d(x, q_{path})$.
    Elegir la dirección del contorno que continua en la misma dirección a la mayoría de los movimientos en dirección a la meta.
  repeat
    Actualizar continuamente d_{reach}, $d_{followed}$ y $[O_i]$.
    Se mueve continuamente hacia $n \epsilon [O_i]$ que es en la dirección elegida.
  until
    q_{goal} es alcanzado **or**
    • Se alcanza el objetivo **or**
    • El robot completa un circulo al rededor del obstáculo, en cuyo caso no se puede alcanzar el objetivo.
    • $d_{reach} < d_{followed}$.
end while

2.5.4. Grafos de visibilidad

Estos métodos se basan en la construcción de un grafo que representa en sus nodos configuraciones factibles del robot, y en sus arcos el coste de conexión directa entre dos vértices. Una vez construido dicho grafo, denominado grafo de conectividad, se halla un camino entre el origen y el destino se reduce a conectar dichos puntos con el grafo y hallar una secuencia de nodos dentro de él [78]. El término de visibilidad alude a que en este método se enlazan los vértices entre los que existe visión directa, es decir, sin obstáculos que lo impidan.

El proceso comienza generando una lista de los vértices de los polígonos a los que se añaden las posiciones inicial y final [79]. Con estos puntos se genera una matriz cuadrada con tantas filas como puntos. A continuación se evalúa para cada

par de puntos su posibilidad de enlace directo, es decir, si el robot ubicado en uno de los puntos puede alcanzar el otro desplazándose en línea recta. Si un obstáculo, en la matriz se anotará un valor que indique tal eventualidad. Esta matriz no es más que la descripción de un grafo que une los punto entre los que es posible el enlace directo (ver figura 21).

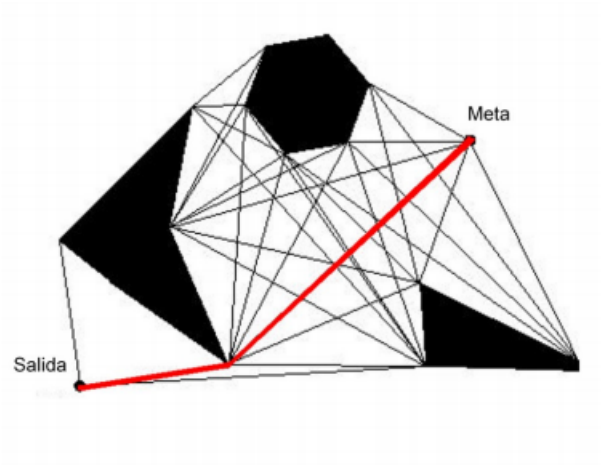


Figura 21: Grafo de visibilidad [80]

Recorriendo este grafo se puede alcanzar cualquier vértice. Así pues, una solución se consigue escogiendo una sucesión de vértices conexos dentro del grafo en la que el primero sea el punto de partida y el último la meta. Entre los muchos caminos a escoger, se suele utilizar el criterio de mínima distancia.

El algoritmo de Dijkstra es el más utilizado entre los grafos de visibilidad, ya que resuelve el problema desde un único vértice origen hasta todos los otros vértices del grafo.

2.5.4.1 Algoritmo de Dijkstra

El algoritmo de Dijkstra también es conocido como algoritmo de caminos mínimos; el objetivo de este algoritmo es determinar el camino más corto desde el nodo de origen al nodo final. Para ello se dota a los nodos de un peso, que equivale a la distancia que hay que recorrer para poder ser alcanzados, y el camino más corto será aquel que la suma de los pesos que lo conforman sea el menor.

Este algoritmo debe su nombre a Edsger Dijkstra, quien lo describió por primera vez en 1959.

Este algoritmo es de tipo greddy, es decir aquel que para resolver un determinado problema sigue la forma heurística (elige la opción óptima en cada paso local

sin considerar opciones futuras). No obstante, el óptimo encontrado en una etapa puede modificarse posteriormente si surge alguna solución mejor.

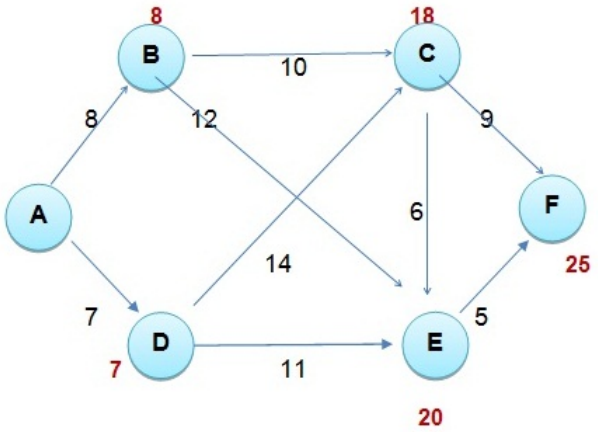


Figura 22: Algoritmo de Dijkstra

Como se puede observar en la imagen 22, el número que aparece sobre cada una de las flechas es la distancia entre los nodos que ésta une, por ejemplo para llegar de A a E hay distintas opciones. A simple vista se podría determinar que las dos más óptimas sería ir de A a E pasando por B (cuya suma de distancia total será 20); o ir de A a E pasando por D cuya distancia total será de 18. Por tanto la ruta más óptima siguiendo el algoritmo de Dijkstra sería pasar por D.

A continuación se implementa el pseudocódigo explicativo de este algoritmo. Para poder desarrollar este algoritmo es necesario crear una un vector (de tamaño n, siendo n el número de nodos existentes) para guardar las distancias del nodo salida al resto (**s**) y se iniciará el vector con distancias iniciales. Además es necesario un vector booleano para controlar los vértices de los que ya se tiene la distancia mínima. El grafo es **G**

Algorithm 4 Dijkstra Algorithm

```
Variables:
entero distancia[n]
booleano visto[n]
  while W ϵ V[G] do
     if no existe arista entre s y w then
           distancia[w] = ∞
     else
           distancia[w] = peso (s, w)
  end while
  distancia[s] = 0
  visto[s] = TRUE
  while NOT visto[s] do
        nodo = coger el mínimo del vector distancia que no esté visto
        visto[nodo] = TRUE
     while w ϵ sucesores (G, nodo) do
        if distancia[w] ¿distancia[nodo] + peso (nodo, w) then
                 distancia[w] = distancia[nodo] + peso(nodo, w)
        end if
     end while
  end while
```

2.5.5. Diagramas de Voronoi

Los diagramas de Voronoi son en realidad los grafos que dan el nombre al método. En este caso el grafo resultante tiene la ventaja de que maximiza la distancia entre el robot y los obstáculos. Este método se utiliza bien directamente [84] o bien como base a otros métodos [85]. La aplicación de este método exige el modelado de los obstáculos como obstáculos C poligonales, lo cual permite adoptar la analogía del robot puntual.

Un diagrama de Voronoi generalizado representa los puntos que equidistan de elementos vecinos entre sí. Estos elementos pueden ser obstáculos o bien bordes del escenario. El atributo de vecindad quiere expresar que no hay ningún otro elemento entre dos que sean vecinos.

Dado que el lugar geométrico de los puntos del plano que equidistan de un punto y una recta es una parábola, y el lugar geométrico de los puntos del plano que equidistan de dos rectas es a su vez una recta, si los obstáculos son polígonos, el diagrama estará compuesto de una sucesión de parábolas y segmentos rectos interconectados. Si se toman los puntos de interconexión como vértices, y los fragmentos de parábolas y rectas como arcos, se dispone de un grafo.

No obstante, los diagramas de Voronoi también pueden extraerse de escena-

rios donde existan obstáculos con lados curvos, aunque en estos casos la forma del diagrama dependerá de estos elementos sin poder acudir a estructuras predefinidas.

Una vez generado el grafo, se procede a conectar los puntos de partida y meta al mismo [79]. Si el punto se halla en la vecindad de dos segmentos, se halla la línea ortogonal que une dicho punto al segmento más próximo. El segmento de dicha recta definido por su intersección con el grafo y el punto considerado, constituirá la unión de este punto al grafo. Si por el contrario el punto se encuentra en la vecindad de un vértice y otro elemento, se halla la recta que pasa por el vértice y el punto, determinándose de la misma forma que en el caso anterior el arco del grafo necesario (ver figura 23). A partir de aquí el problema resulta como en el caso de los grafos de visibilidad, es decir, se podría aplicar el algoritmo de Dijkstra hallando así una secuencia de vértices en el grafo que una sendos puntos.

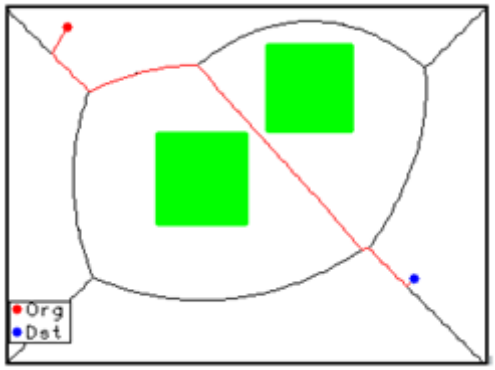


Figura 23: Diagrama de Voronoi [82]. El punto de origen es de color rojo y la meta, azul.

La ventaja de este método es la capacidad de alejarse de los obstáculos, eligiendo siempre el camino más despejado posible. Este método es conveniente en escenarios con abundantes obstáculos y pasos estrechos, sin embargo resulta muy inconveniente en entornos con escasos obstáculos. En este caso, dependiendo de la ubicación de los mismos, el diagrama arrojará trayectorias que pueden dar rodeos largos e innecesarios para llegar a la meta.

2.5.6. Algoritmos basados en celdas

En esta sección se explicarán otro tipo de representación del espacio libre como es la descomposición en celdas de éste. Esta estructura representa el espacio libre mediante la unión de las regiones más simples, conocidas como celdas.

Dos celdas son adyacentes si ambas tienen parte del contorno del objeto; una gráfica adyacente codifica la relación entre las diferentes celdas, donde un nodo

corresponde a una celda y el borde conecta el los nodos de las celdas adyacentes.

La idea general de estos métodos consiste en dividir el problema en dos niveles: uno global que resuelve la trayectoria por zonas y otro, más simple, a nivel local. Se trata de dividir el entorno de estudio en un conjunto de celdas [78]. Éstas han de tener la peculiaridad de que resulte sencilla la conexión de dos puntos cualesquiera de su interior, bien porque sean muy próximos, porque la celda sea convexa, etc.

Una vez discretizado el espacio de configuraciones libres de colisión en un conjunto de celdas completo se procede a resolver su conectividad a un nivel superior construyendo un grafo de conectividad.

Existen dos subtipos principales que se estudiarán a continuación: la descomposición de celdas exactas y la descomposición aproximada.

2.5.6.1 Descomposición en celdas exactas

Se utiliza este término cuando la descomposición genera un conjunto de celdas cuya unión resulta idéntica al espacio de configuraciones libres de colisión. Es decir, las celdas han de estar de tal modo definidas que se adapten a la configuración de obstáculos.

Por ejemplo, en un problema en el que el escenario y los obstáculos tengan limites poligonales, como se muestra en la imagen 24, se pueden trazar rectas verticales en cada vértice. De esta forma, todo el espacio de configuraciones libres de colisión queda fragmentado en celdas trapezoidales o triangulares en su totalidad. A partir de aquí se genera el grafo de conectividad y se halla la secuencia de celdas en donde ha de estar contenida la trayectoria.

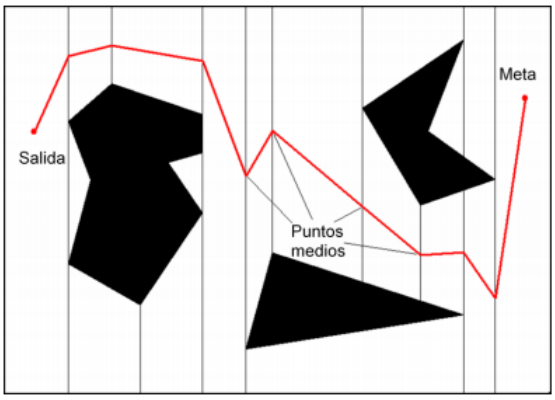


Figura 24: Descomposición en celdas exactas [80]

Para generar la trayectoria basta con elegir los puntos medios de los segmentos adyacentes como puntos de paso. Trazando segmentos rectos entre los mismos se

obtiene la solución.

Este tipo de fragmentación se denomina **Descomposición Vertical** [83]. Precisa una ordenación previa de los vértices por abscisa, lo cual implica un tiempo de cómputo $O(n \log n)$, siendo n el número de vértices.

A partir de aquí, el trazado de líneas verticales, la designación de celdas y la identificación de las que contienen la posición origen y la de destino; todo puede realizarse de modo concurrente en un tiempo $O(n \log n)$. El número de celdas y el de sus enlaces coinciden en $O(n)$.

Para un grafo de estas dimensiones, hallar un camino entre dos nodos puede realizarse en un tiempo $O(n)$ aunque no sería el más corto. si se desea esta última propiedad el algoritmo de búsqueda tardaría $O(n \log n)$ [78].

Otro ejemplo de descomposición en celdas exactas lo constituye la llamada **triangulación de Delaunay**. Ésta considera inicialmente el conjunto de puntos formado por los vértices de los obstáculos. A continuación une en un segmento los pares de puntos tales que es posible hallar una circunferencia que los contenga sin inscribir ningún otro punto. Esta exclusión implica que los puntos del par son los más cercanos al centro de la circunferencia hallada. Además, dicho centro equidista de los puntos a unir, puesto que para dos puntos cualesquiera de una circunferencia, la bisectriz del segmento que definen contiene siempre al centro de la misma. De ambos resultados se deduce que este centro pertenecerá al diagrama de Voronoi que definen los vértices de los obstáculos.

No obstante, el método no resuelve el diagrama de Voronoi, sino que únicamente define una descomposición de C_{free} en celdas triangulares delimitadas por los segmentos definidos con la propiedad mencionada. sobre estas celdas se construye un grafo de adyacencia y se resuelve.

Este método es más complejo que la descomposición vertical; no obstante, los puntos medios de los segmentos de estas celdas se distancian con mayor eficacia de los obstáculos, y la solución obtenida ofrece por tanto mayor margen de seguridad en orden a evitar colisiones, como se puede apreciar en la imagen 25 .

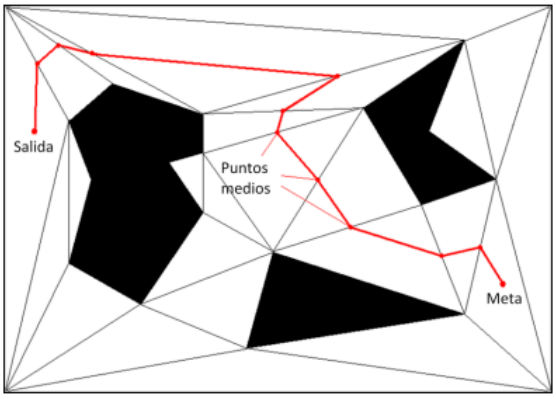


Figura 25: Descomposición de Delaunay [80]

2.5.6.2 Descomposición en celdas aproximadas

En este tipo de descomposición se utilizan celdas de diseño predefinido, habitualmente rectangulares. Al descomponer todo el escenario en dichas celdas algunas solaparán bien completamente bien en parte algún obstáculo. Éstas serán descartadas, las demás serán las que conformen el grafo de adyacencia. Por lo demás el proceso es bastante similar. La ventaja de este método con respecto a la descomposición exacta reside en su rapidez de implementación.

Como puede deducirse este tipo de descomposición no cumple con el requisito de la anterior debido a esas celdas descartadas que contienen en parte configuraciones libres de colisión. A continuación se describen dos ejemplos: el método de división en cuadrantes, también conocido como descomposición *quadtree* y el método de frente de ondas, aludido también como *fast marching method*.

El método de descomposición ***quadtree*** parte de un escenario contenido en un rectángulo, a continuación se divide el escenario en rectángulos semejantes mediante el procedimiento de trazar mediatrices en la base y la altura del original (como se observa en la imagen 26), obteniendo así 4 rectángulos. Se clasifican los triángulos según su área esté contenida en CB (se les llamará "llenos"), en C_{free} (rectángulos "vacíos"), o tenga parte en ambos ("mixtos").

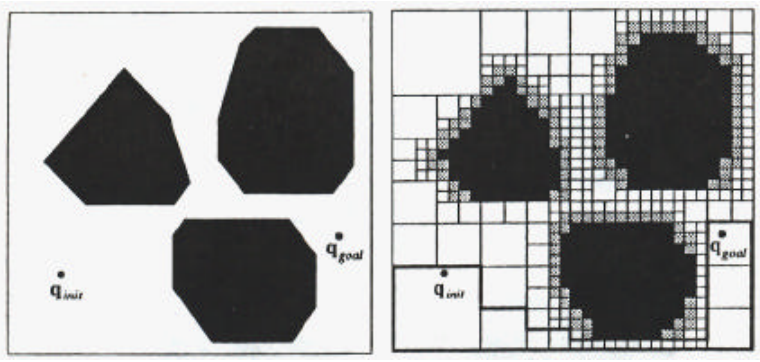


Figura 26: Descomposición en celdas aproximadas: Quadtree [12]

Los rectángulos mixtos van volverán a sufrir el proceso de fragmentación utilizando el mismo sistema. De nuevo serán etiquetados como llenos (si están dentro de CB), vacíos (si están en C_{free}) y mixtos, siendo los mixtos los sujetos del siguiente proceso.

En cada una de estas etapas se actualiza una estructura de datos de tipo árbol de grado 4, es decir, de cada nodo del árbol descienden a otros cuatro. El nodo raíz representa el rectángulo inicial que contiene el escenario. Sus cuatro nodos descendientes representan los rectángulos de la primera subdivisión. De la misma forma está relacionados los demás rectángulos, teniendo en cuenta que aquellos nodos que están etiquetados como llenos o vacíos no tienen nodos descendientes, solo los mixtos los tienen.

En la figura 27 se ha representado un ejemplo de este árbol marcando en gris los nodos mixtos, en blanco los vacíos y en negro los llenos. Utilizando este árbol, se comprueba en cada iteración si existe una secuencia de celdas vacías que conecte la celda origen con la celda destino. Si la hay, se procede a generar un camino utilizando los puntos medios de los rectángulos vacíos. Si no la hay, se vuelve a iterar. Estas iteraciones están limitadas a un determinado tamaño de celda, con lo que si se alcanza sin encontrar un camino, se devuelve que no ha sido posible hallar una solución.

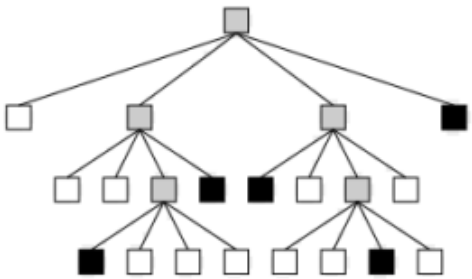


Figura 27: Estructura en árbol de las particiones [80]

Este método es adecuado para el procesamiento de escenarios definidos como imágenes, donde cada pixel pertenecerá o bien a C_{free} o bien a CB. En este caso se define la celda mínima con el tamaño de un pixel. De este modo, al llegar a ese nivel de descomposición, no quedarán celdas mixtas y el método se detiene de forma natural.

Este método ha sido diseñado en un espacio de dos dimensiones, pero puede ser generalizado a varias. En el caso de tres dimensiones se habla de descomposición *octree*, debido a que el árbol pasa a ser de grado 8.

En cambio, existen otros métodos que directamente parten de una descomposición en celdas del mismo tamaño, es decir, utilizan una discretización regular del espacio de configuraciones. Uno de estos métodos es conocido como ***fast marching method*** [91]. Se basa en descubrir la trayectoria de un rayo de luz emitido desde la posición origen hasta la de destino, aplicando teoría de ondas.

2.5.7. Modelado del espacio libre

En este método, los obstáculos se les representa como polígonos. La planificación se lleva a cabo a través de los CRG, cilindros rectilíneos generalizados, y al igual que Voronoi, con el uso de los CRG se pretende que el robot se mueva lo más alejado de los obstáculos.

La ruta será una configuración de CRG interconectados, tal que la configuración inicial o de partida se encuentre en el primer cilindro de la sucesión y la configuración final en el último cilindro.

El proceso para construir un CRG será el siguiente:

1. Cálculo del eje del CRG, se define como la bisectriz del ángulo α formado por el corte de las rectas que contienen las aristas ${}^1a_i$ y ${}^2a_j$ que cumplen con las condiciones antes mencionadas.

2. Por ambos lados de dichas aristas se construyen rectas paralelas al eje, con origen en los vértices de las aristas implicadas y con extremo señalado por la proyección del primer obstáculo que corta el eje.

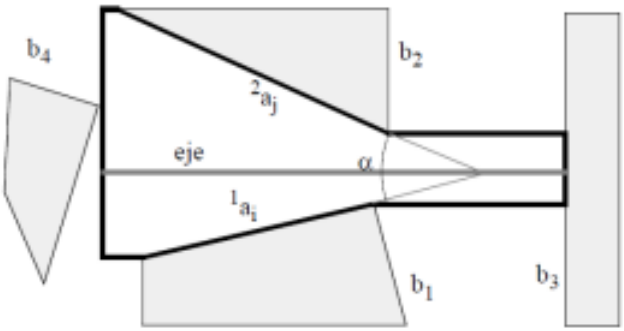


Figura 28: Construcción de un CRG [86]

Repitiendo este proceso se construye una red CRG en el entorno del robot que modela el espacio libre del mismo. El robot navegará por el eje del cilindro, en el cual se encuentran anotadas para cada punto el rango de orientaciones admisibles.

El paso de un CRG a otro se produce siempre y cuando sus ejes intercepten y la intersección del rango de orientaciones admisibles en el punto de corte de ambos ejes no sea nulo [86].

3. Descripción general

Teniendo en cuenta todo lo mencionado anteriormente se puede comenzar a acotar la parte práctica proyecto.

El trabajo consiste en generar una trayectoria para un robot submarino. En este caso el sistema tiene que ser capaz de ubicarse y ejecutar una trayectoria para alcanzar la posición final definida sin colisionar con ningún obstáculo que se encuentre en el recorrido.

Para resolver este problema, por tanto se utilizarán distintos simuladores, que facilitarán la ejecución de las trayectorias y la evitación de obstáculos, como son *UWSim*), el asistente *MoveIt* y la herramienta de visualización 3D *Rviz*. Todos estos simuladores son explicados en detalle en la sección 5.

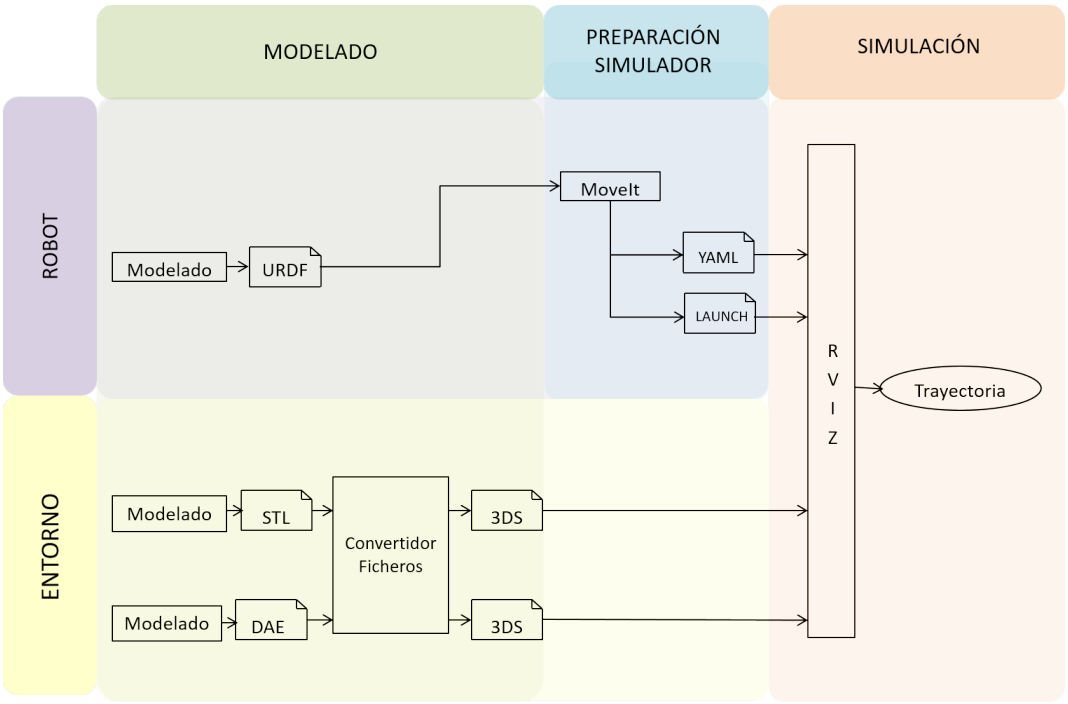


Figura 29: Arquitectura general del proyecto

En la imagen 29 se muestra el proceso que habría que seguir desde el diseño del robot y del entorno, hasta la generación de trayectorias. Este proceso se explicará posteriormente en la sección 7.

Por tanto, este sistema estará formado por varios componentes que pertenecerán a distintas categorías dentro del trabajo en sí y se dividirá en 4 grandes bloques:

- **Hardware.** No es la parte en la que se centra el proyecto, es cierto que sin el equipo empleado no se podría haber llevado a cabo.

- **Software.** *ROS* es una plataforma de código abierto en la que las aplicaciones se comunican entre sí, emitiendo y recibiendo información. Además, en el desarrollo de la parte práctica se utilizarán otras herramientas, no solo relacionadas con *ROS*.

- **Generación de la trayectoria.** Engloba todo lo que se ha llevado a cabo para la realización de la parte práctica del proyecto, es decir, como se integra en el sistema y los resultados que se obtienen.

En función del número de grados de libertad controlables se utilizarán distintos algoritmos para alcanzar el objetivo final.

4. Hardware

Pese a que el proyecto se encuentra centrado en un submarino (GIRONA 500), el trabajo se desarrolla en su totalidad fuera de éste, ya que el AUV con el que se ha trabajado se encuentra en la Universidad de Gerona y no es posible utilizarlo en la comprobación de resultados.

Por tanto se trabajará en su totalidad a través de un ordenador para generar la trayectoria y visualizarla en el entorno de *ROS*.

4.1. Portátil HP Pavilion

Las restricciones que se pueden encontrar en este caso son el ser capaz de trabajar con *ROS* cuyas exigencias no son excesivas, y por extensión poder trabajar con Ubuntu 14.04. En vez de realizar una partición del disco, era más sencillo instalar el sistema operativo en un disco duro, ya que esto permitiría trabajar en el proyecto desde cualquier ordenador, sin necesidad de tener físicamente el ordenador en el que se está trabajando.

Las prestaciones que caracterizan el modelo y se encuentran relacionadas con los requerimientos tanto del sistema operativo como de *ROS* son:

- **Procesador:** AMD A10-5745M APU con Radeon
- **Memoria:** 500GB
- **Tarjeta Gráfica:** 2.1 GHz

Figura 30: Portátil HP Pavilion

4.2. Disco duro

Se ha empleado un disco duro de 500Gb en el que se instalará *Ubuntu* y todos los programas que se necesiten para la elaboración del proyecto.

5. Software

El objetivo principal de este trabajo es simular un AUV que permita alcanzar una posición final sin colisionar con ninguno de los obstáculos que se encuentren en su trayectoria.

Por un lado se encuentra *ROS* (el entorno de trabajo), es la parte que caracteriza el proyecto debido a su particular diseño modular. Por otra parte, se encuentra el simulador *UWSim*, gracias a él se obtiene la descripción del robot que se va a utilizar para la simulación. Además, para realizar las trayectorias de una manera más intuitiva se empleará un simulador de entornos, conocido como *Rviz* y un asistente, *MoveIt*, que ayudará a crear el paquete, y por tanto, los ficheros necesarios para poder visualizar tanto el entorno como el robot en *Rviz*.

Antes de comenzar a explicar cada herramienta o simulador utilizado, es necesario tener una visión general de sobre que sistema operativo o herramienta va instalado cada uno de ellos.

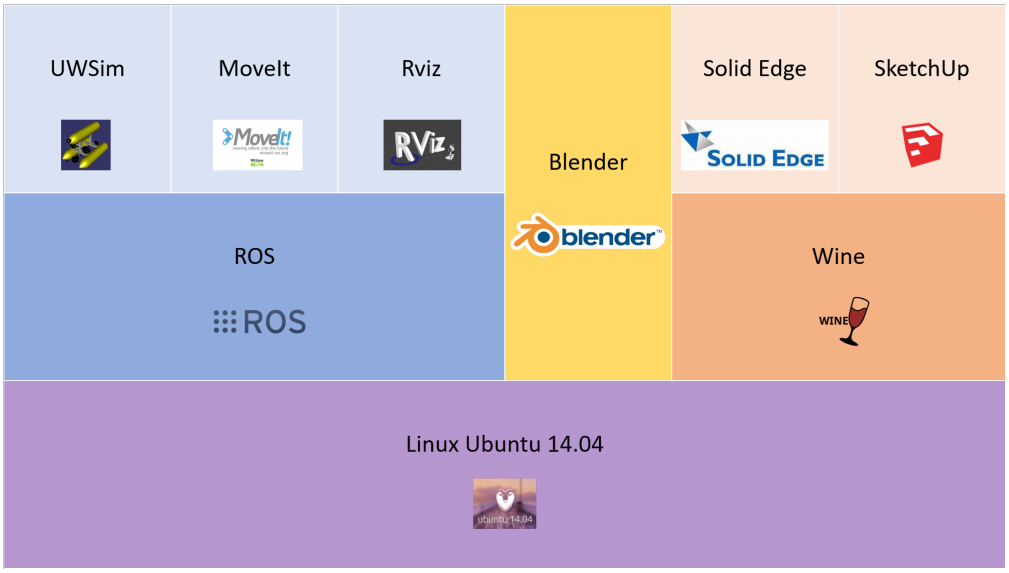


Figura 31: Software stack

Como se observa en la imagen 31 siempre se ha trabajado sobre el sistema operativo *Ubuntu 14.04*. *ROS* va instalado directamente sobre el sistema operativo, al igual que *Blender* y *Wine* (permite la ejecución de programas diseñados para *Windows*). Una vez instalado *ROS*, sobre él se instalan el resto de simuladores y asistentes relacionados con el robot, como son *UWSim*, *MoveIt* y *Rviz*; mientras que *Solid Edge* y *SketchUp* están instalados sobre *Wine*, ya que no existe una versión para *Ubuntu* de estos programas.

5.1. Entorno de Programación: ROS

Se ha estudiado el funcionamiento y sensores del AUV GIRONA 500 y su forma de uso bajo distintos simuladores, para todos ellos ha sido necesaria la integración del *framework ROS* (*Robot Operating System*) para las comunicaciones entre el componente y el robot. Esta tecnología permite enviar órdenes al robot a través de código desarrollado en Python o en C++.

ROS es un entorno de trabajo creado para diseñar y ejecutar aplicaciones de robótica. Es un conjunto de herramientas, bibliotecas y convenciones que buscan simplificar la tarea de crear conductas para robots complejas y robustas a través de una amplia variedad de plataformas relacionadas con la robótica [33].

Figura 32: Logotipo *ROS* Indigo [32]

ROS fue desarrollado originalmente en 2007 con el nombre de *switchyard* por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto que ellos estaban llevando a cabo allí (Robot con Inteligencia Artificial). Desde 2008 el desarrollo continua en un instituto de investigación robótico con más de veinte instituciones colaborando.

Como no se había realizado ningún trabajo con este entorno de programación, antes de comenzar este proyecto se realizaron una serie de tutoriales que provee la página oficial de *ROS* [30].

5.1.1. Conceptos Principales

A nivel conceptual, el sistema de construcción de *ROS* en su versión *Indigo, catkin*, se divide en tres niveles diferentes: Sistema de Archivos, Gráfica de Computación (*Computational Graph*) y Comunidad [31]. Ya que los dos primeros niveles

intervienen directamente con la comprensión del trabajo como conjunto, se pasa a describirlos con mayor detalle.

5.1.1.1 Sistema de Archivos

Los conceptos relacionados con el Sistema de Archivos cubre los recursos que se encuentran en el Disco Duro relacionados con *ROS* [31].

- **Paquetes (*package*):** Es la unidad principal de organización de software con la que trabaja *ROS*. Un paquete puede contener aplicaciones denominadas nodos, bibliotecas con dependencia a *ROS*, conjuntos de datos, ficheros de configuración, ficheros ejecutables o cualquier cosa que sea práctico organizar junto a lo anterior. Los paquetes son la unidad atómica de construcción y liberación presente en *ROS*.

- **Manifiestos de paquetes (*package manifest*):** El fichero *package.xml* presente en un paquete, proporciona información relacionada con el paquete en el que se encuentra contenido (como el nombre del fichero, la descripción, la información relacionada con su licencia o sus dependencias con respecto a otros paquetes entre otros).

- **Repositorios (*repositories*):** Los repositorios son una colección de paquetes que comparten un sistema *VCS* (*Version Control System*) común. Los paquetes que comparten un *VCS* comparten la misma versión y pueden ser liberados al mismo tiempo.

- **Tipos de Mensajes (*msg*):** Las descripciones de los mensajes definen como se encuentra estructurada la información contenida en un mensaje que se envía en *ROS*. Esta información se almacena en una dirección similar a *my_package/msg/MyMessageType.msg*.

- **Tipos de servicios (*srv*):** Las descripciones de los servicios definen las estructuras de datos de petición y respuesta para servicios en *ROS*. Estos servicios se guardan en la dirección *my_package/srv/MyServiceType.srv*.

5.1.1.2 Gráfica de Computación

La gráfica de computación se define como la red *peer-to-peer* (todos los pequeños programas que pueden formar un sistema *ROS* se comunican mediante

mensajes que circulan de uno a otro sin pasar por una rutina de servicios central).

Los conceptos relacionados con este ámbito con mayor relevancia son los nodos (*nodes*), el Master (*Maester*), el Servidor de Parámetros (*Parameter Server*, mensajes (*messages*), servicios (*services*), tópicos (*topics*), y bolsas (*bags*). Todos ellos proporcionan información a la Gráfica de diferentes maneras:

- **Nodos (*nodes*):** Son ejecutables que realizan diversos procesos. *ROS* está diseñado para ser modular a la mayor escala posible; por lo que el sistema de control de un robot conjugará varios nodos.

- **Máster (*Master*):** El *ROS Master* proporciona el registro y búsqueda de nombres al resto de la Gráfica de Computación. El Máster permite que los nodos se encuentren unos a otros, intercambien mensajes o pidan servicios.

  El Máster *ROS* actúa como un servicio de nomenclatura en la Gráfica de Computación. Almacena tópicos y provee de datos de registro para los nodos. Los nodos se comunican con el Máster para reportar acerca de sus datos de registro. Al comunicarse con el Máster reciben también información acerca de otros nodos registrados, lo que permite establecer comunicaciones correctamente. El Máster también permite informar a los nodos cuando los datos de registro cambien, de modo que los nodos que ya se encuentran funcionando puedan establecer comunicaciones cuando un nuevo nodo se ejecuta.

  Aun así los nodos se conectan entre sí directamente, el Máster únicamente se encarga de darles información de búsqueda, de un modo parecido a como lo haría un servidor *DNS*. Los nodos que se subscriben a un tema pedirán conexión a los nodos que la publican en dicho tema, y establecerán una conexión a través de protocolo. El protocolo de conexión más empleado en *ROS* es el *TCPROS*, basado en protocolos *TCP/IP* estándar.

- **Servidor de Parámetros (*parameter server*):** El Servidor de Parámetros permite guardar información bajo llave en una localización principal. Actualmente forma parte del Master.

- **Mensajes (*messages*):** Los nodos se comunican entre sí intercambiando mensajes. Un mensaje es una estructura de datos simple, que almacena una serie de variables (las variables estándar como *integers*, *floats* o *bools* están incluidas) u objetos de una clase.

- **Tópicos (*topics*):** Los mensajes son enviados empleando semántica de tipo publicación/suscripción. Un nodo envía un mensaje publicándolo en un determinado *topic* o tópico. Un tópico es la denominación que se emplea para identificar el contenido del mensaje. Un nodo que esté interesado en información de un determinado tipo, se suscribirá al tópico correspondiente.

  Para un solo tópico puede haber múltiples publicadores y suscriptores, así como un solo nodo puede publicar y suscribirse a varios temas la mismo tiempo. Siguiendo la idea de separar la consumición de información de su emisión los nodos publicadores no son conscientes de la existencia de los suscriptores y viceversa.

- **Servicios (*services*):** A pesar de que el sistema de publicación y suscripción constituye un ejemplo de comunicación muy flexible, no resulta del todo apropiado cuando se busca una interacción de tipo petición/respuesta, normalmente requeridos en sistemas distribuidos.

  En *ROS*, esta interacción se realiza a través de *services* o servicios, que se definen empleando un par de estructuras de mensajes: una se emplea para la petición y otra para la respuesta correspondiente. Un nodo ofrece un determinado servicio bajo un nombre y un cliente emplea dicho servicio enviando un mensaje de petición y esperando una respuesta. La mayor parte de bibliotecas de *ROS* ofrecen esta interacción con el programador como si se tratara de una llamada a un procedimiento remoto.

- **Bolsas (*bags*):** Los *bags* o bolsas son un formato dedicado a guardar y reproducir información relacionada con mensajes en *ROS*. Los *bags* son un mecanismo de gran relevancia para almacenar información, como por ejemplo información de un sensor, que suele ser relativamente difícil de guardar pero indispensable a la hora de desarrollar y probar algoritmos.

Este tipo de arquitectura permite operaciones desvinculadas, donde los nombres son los medios primarios a través de los cuales se pueden construir sistemas mayores y más complejos. Los nombres tienen una gran importancia dentro de *ROS*, ya que los nodos, tópicos, servicios y parámetros poseen nombres. Cada biblioteca de cliente de *ROS* soporta el remapeado de nombre a través de la línea de comando, lo que implica que un programa que se encuentra compilado puede ser reconfigurado mientras que se está ejecutando para trabajar en una topología diferente en la Gráfica de Computación.

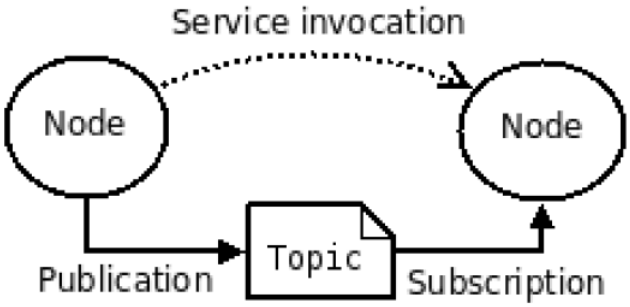


Figura 33: Esquemático de la Gráfica de Computación [31]

5.1.1.3 Comunidad

Durante los últimos años, la comunidad de *ROS* ha crecido considerablemente. Históricamente, los usuarios lo utilizaban solo en los laboratorios, pero en la actualidad, cada vez es más frecuente encontrar empresas que lo utilizan en la industria, debido a la automatización.

La comunidad de *ROS* es muy activa, ya que tiene más de 1500 participantes en la lista de emails de *ROS*, 3300 que trabajan en colaboración con la documentación de la Wiki (páginas web con tutoriales de como utilizar las distintas herramientas y simuladores) y 5700 participantes que preguntan en los foros de *ROS*.

5.1.2. Componentes básicos de *ROS*

5.1.2.1 Servicio *roscore*

Roscore es un servicio que *ROS* ofrece permitiendo así una comunicación entre nodos mediante el envío de mensajes. Cualquier aplicación de *ROS* necesita que esté abierto *roscore* en la terminal.

Cada nodo se conecta al *roscore* en cuanto se inicia y le envía los detalles de los mensajes que desea enviar y a los que se quiere suscribir, también establece las conexiones *peer-to-peer* con otros nodos cuando uno nuevo aparece.[34]

Cada nodo informa al *roscore* de los mensajes a los que quiere suscribirse y qué mensajes envía y es entonces cuando *roscore* informa de las direcciones a aquellos interesados en dichos mensajes.

Roscore también provee de un servidor de parámetros que es utilizado por los nodos para su configuración. Permite a los nodos almacenar y recuperar datos arbitrarios. Para esto, como para la mayoría de servicios de *ROS*, existe el comando *rosparam*.

5.1.2.2 *Catkin* y *workspaces*

Para poder trabajar con *ROS*, además de todos los conceptos explicados anteriormente, es importante conocer los espacios de trabajo en los que se desarrollarán los programas.

Catkin es el sistema de construcción de *ROS* desde la versión *Groovy* (ya que antes se utilizaba el comando *rosbuild*), que mediante la combinación de macros *CMake* y *scripts* en lenguaje *Phython*, nos permite la generación de *targets* a partir de puro código y permite que pueda ser utilizado por un usuario final. Los *targets* son librerías, programas ejecutables o interfaces. Estos códigos se agrupan por funcionalidades comunes en los ya mencionados paquetes.

Para construir estos paquetes, el sistema de construcción necesita información como la localización de los componentes de la cadena de herramientas, la localización de los códigos fuente, las dependencias tanto de los códigos como externas, donde se encuentran esas dependencias, los *targets* que se deben construir, etc. Toda esta información es la que con el sistema *CMake*, se encuentra en los ficheros denominados *CMakeLists.txt*.

Catkin es un sistema de construcción a medida que permite que *CMake* maneje las dependencias entre paquetes.

Esto es necesario para que *ROS*, ya que éste está formado por un conjunto de paquetes libremente federados, por lo que podemos tener muchas dependencias entre paquetes a priori independientes que además utilizan diferentes lenguajes de programación. Por esto, el proceso de construcción de un *target* puede variar mucho entre sí, incluso dentro de un mismo paquete. El objetivo de *catkin* es facilitar la construcción y el funcionamiento de código *ROS* mediante la utilización de herramientas y convenciones para simplificar el proceso. Esta herramienta es la que se basa la construcción de los espacios de trabajo (*workspaces*).

Antes de realizar cualquier programa en *ROS* es necesario crear un *workspace* donde se guardará y se trabajará con el código. Un *workspace* simplemente es un conjunto de directorios en los que una agrupación de códigos es almacenada. Se puede tener dentro de una máquina una gran cantidad de *workspaces*, y pueden encontrarse todos en los tutoriales que la propia página que *ROS* ofrece [30], sin embargo y para mayor comodidad, se ha adjuntado una guía rápida de instalación (basada en estos tutoriales) en el anexo A.1 de este documento.

El comando *catkin_make* genera dentro de *workspace* dos nuevos directorios: *build* y *devel*. El primero es donde se almacenan los resultados de la ejecución del *catkin*, como pueden ser ejecutables o librerías, mientras que en el segundo hay una gran cantidad de ficheros y directorios de configuración del *workspace* [35].

5.1.2.3 Paquetes

Una vez conocidos los *workspaces*, se profundizará en el concepto de paquete de *ROS*. Como ya se ha comentado anteriormente, un paquete es la unidad fundamental de agrupar la información en *ROS*. Estos paquetes se almacenan dentro del directorio *src* que se ha credo previamente en el *workspace* y es imprescindible que contengan tanto un fichero *CMakeList.txt* como un *package.xml*. Estos archivos describen al comando *catkin* cómo debe interactuar con ellos cuando es ejecutado.

En estos paquetes es donde se almacenarían los programas realizados en otros lenguajes de programación y los ficheros que son necesarios para que la funcionalidad del paquete pueda ejecutarse correctamente.

5.1.3. Comandos importantes de *ROS*

Los comandos más importantes ya se han explicado con anterioridad.

```
roscore
```

El comando *roscore* es el primer comando que se ha de lanzar a la hora de ejecutar cualquier aplicación de *ROS* y su importancia dentro de cualquier proceso de *ROS*, pero no es la única instrucción que se utilizará con frecuencia. A continuación se mencionan los de mayor importancia.

```
rosrun
```

Rosrun es un comando que permite lanzar los programas sin que exista la necesidad de buscar la ubicación de los mismos dentro del sistema de archivos de *Ubuntu*. Únicamente utiliza el nombre del paquete y del programa concreto que se desea ejecutar, sin importar dónde se encuentre la terminal situada dentro del sistema de ficheros. La única condición que se ha de cumplir previamente es que se haya ejecutado el comando *roscore* y ésta esté activo.

```
roslaunch
```

Roslaunch se trata de un comando que permite lanzar varios nodos de forma simultánea. A la hora de ejecutarse, se hace de forma muy simular al *rosrun*, pero en este caso se necesita un tipo de ficheros especiales con extensión *.launch*. Son ficheros *XML* que almacenan toda la información necesaria de los nodos que se quiere lanzar.

Existen otros comandos también necesarios para realizar todas las acciones que se desean y permiten navegar por el sistema de archivos de *ROS*.

- *rospack*: permite recibir información sobre los distintos paquetes que se encuentran en la máquina empleada.

- *roscd*: este comando es equivalente a *cd* para *ROS*, ya que permite la navegación por las distintas carpetas o paquetes desde la terminal de *Ubuntu*.

- *rospd*: similar al comando anterior, pero éste recuerda la última ubicación en la que se estaba trabajando.

- *rosls*: proporciona una lista de los paquetes que se pueden encontrar en el directorio en el que se está trabajando.

- *rosed*: permite modificar un fichero que se encuentra dentro de un determinado paquete desde la terminal.

- *roscp*: se emplea para copiar un fichero de un paquete a otro.

- *rosnode*: proporciona información sobre un nodo.

- *rostopic*: permite obtener información sobre un tópico.

Estos comandos se encuentran dentro de un paquete que está incorporado a *ROS*, llamado *rosbash*, y que además permite la opción de autocompletar cuando se presiona el tabulador.

Simuladores

Los simuladores son una parte clave de la robótica. Permiten poder depurar los códigos y los modelos matemáticos antes de implementarlos en el robot, pudiendo así evitar ciertos errores que pueden provocar que la vida útil del robot quede reducida muy considerablemente. Además, facilita la tarea de elegir un robot para una función determinada sin necesidad de hacer ningún tipo de inversión previa y con unos errores mínimos. A continuación se exponen todos los simuladores y herramientas de simulación que se han utilizado para resolver el problema planteado.

5.2. *UWSim*

UWSim (UnderWater Simulation) [45] es un simulador de vehículos submarinos desarrollado para dar soporte a dos proyectos, RAUVI y TRIDENT. En ambos proyectos el objetivo era que el robot recuperase un objeto sin la dirección de ningún operario.

No obstante *UWSim* es sólo un simulador cinemático, por lo que carece de la dinámica necesaria para tener una simulación realista de AUVs y ROVs.

5.2.1. Definición

UWSim es una herramienta de software para la visualización y simulación de los robots submarinos operados mediante control remoto (ROV). Este tipo de robots no requieren de tripulantes pero si de un operador, que vía remota dirige el submarino. Este software visualiza el escenario virtual submarino que puede ser configurado utilizando un software de modelado.

Al vehículo submarino se le pueden añadir controladores, características de la superficie, manipuladores robóticos y sensores simulados entre otros, gracias a las interfaces de red. Esto permite integrar con facilidad la simulación y la herramienta de visualización con las arquitecturas de control permitiendo Hardware In The Loop simulations.

Este simulador ha sido utilizado para simular la logística de las intervenciones submarinas y para reproducir misiones reales.

El software *UWSim* es *open source* (de código abierto, distribuido y desarrollado libremente).

Experimentar con robots submarinos tiene cierta complejidad,ya que se necesitan demasiados recursos, como por ejemplo un tanque lo suficientemente grande

en el que situar el robot submarino, lo cual genera unos costes de mantenimiento demasiado elevados. Otra posibilidad sería probar los robots en espacios abiertos como lagos o el mar; esta opción también supone un coste elevado y requiere logística especial. Ademas supone un problema para los investigadores, ya que no pueden observar la evolución del sistema desde la superficie.

Para facilitar todos estos problemas de desarrollo de robots submarinos es importante desarrollar simuladores que permitan probar los sistemas antes de desarrollarlos y supervisarlos bajo el agua, donde los desarrolladores tiene una visión directa del sistema.

5.2.2. Diseño del Software

UWSim está implementado en C++ y utiliza las librerías *OpenSceneGraph* (OSG) y *osgOcean*, además de añadir funcionalidades para facilitar el incluir robots submarinos, simular sensores y hacen de interfaz con los programas de control externos mediante *ROS*.

5.2.3. Características

Las principales características del simulador se exponen a continuación.

5.2.3.1 Entorno

Las escenas que se muestran en el simulador son modelables e incluyen materiales y texturas. Además de la estructura en 3D, se pueden añadir elementos dinámicamente, se pueden modificar o incluso eliminar desde el programa principal. OSG representa el escenario virtual, donde los nodos son fácilmente accesibles y controlados.

Existen distintos bloques modificables. La escena completa es descrita por el usuario en un fichero XML. Los principales bloques dentro del XML son los que se explican a continuación.

- **OceanState:** permite la configuración de los parámetros del océano, así como la dirección y velocidad del viento (afecta a la cantidad y el tamaño de las olas), el color del agua, la visibilidad y los factores de atenuación.

- **SimParams:** el usuario puede desactivar los efectos de visualización gracias a esta opción. Además permite establecer el origen de coordenadas y establecer un *offset* con respecto al origen definido por defecto.

- **Cámara:** permite definir los parámetros de la cámara principal. Ésta es la que observa la escena y todo lo que sucede en el entorno.

  El usuario puede definir el modo de la cámara y el campo de visión entre otros. También es posible definir los parámetros de la cámara desde la calibración de la matriz.

- **Vehículo:** incluye los robots submarinos. El usuario tiene que especificar la descripción del robot en el fichero *URDF* (Unified Robot Description Format); donde se determinarán las juntas del robot, la posición del mismo y en caso de ser necesarios los sensores.

- **Objeto:** incluye otros modelos 3D a la escena desde los ficheros existentes en cualquiera de los formatos 3D soportados por *OSG*.

- **RosInterfaces:** relaciona las interfaces de *ROS* con algunos objetos. Esto proporciona información de sensor al software externo y además recibe referencias externas, las cuales se utilizan para actualizar la posición de los objetos y los robots en la escena.

5.2.3.2 Soporte de múltiples robots

El diseño del software incluye clases abstractas especializadas para añadir soporte para distintos vehículos submarinos. El vehículo por defecto está compuesto de un modelo en 3D (el modelado se ha realizado en un software aparte) que se puede ubicar dentro del escenario que se desee. Este robot tiene 6 grados de libertad y es el empleado en la generación de trayectorias. El fichero en el que se almacena la información relativa al robot es del tipo XML y en el se incluye información cinemática, dinámica y visual del vehículo visual.

Dentro de la misma escena puede haber dos o más robots que están siendo controlados simultáneamente.

5.2.3.3 Sensores simulados

Para cualquiera de los vehículos hay doce sensores disponibles además de los establecidos por defecto que sirven para determinar la posición y la velocidad. Estos dos últimos datos se definen mediante los 6 grados de libertad: (x, y, z, r, p, y).

Las 3 primeras coordenadas correspondientes con la posición con respecto al origen de coordenadas establecidas y las 3 últimas son los ángulos de navegación usados para describir la orientación de un objeto en tres dimensiones (las letras se corresponden con *Roll*, *Pitch* y *Yaw*, respectivamente).

Los sensores disponibles para cualquier vehículo son los mostrados a continuación.

- **Cámara:** proporciona imágenes virtuales que pueden ser utilizados para el desarrollo de algoritmos de visión.

- **Rango dinámico de la cámara:** permite captar el detalle de la imagen , ya que el sensor capta todos los rangos de tonalidades presentes en la imagen.

  Una escena con un alto rango dinámico tendrá una gran diferencia de brillo entre las zonas más oscuras y las más brillantes.

- **Sensor de alcance:** mide la distancia al objeto más cercano sobre una dirección predefinida, con un rango máximo de 10 metros. Se puede unir al submarino de la misma manera que una cámara.

- **Recolector de objetos:** coge el objeto cuando este se encuentra dentro del la distancia predefinida.

- **Presión:** proporciona la medida de la presión.

- **DVL:** estima la velocidad lineal a la que el vehículo se está moviendo.

- **IMU:** estima la orientación del robot con respecto al origen de coordenadas del mundo.

- **GPS:** proporciona la posición del vehículo con respecto del mundo, pero solo funciona cuando el robot se encuentra cercano a la superficie.

- **Multibeam:** simula un array de sensores de rango y proporciona información sobre las distancias a los objetos más cercanos sobre el eje Z de su sistema de coordenadas local variando la rotación en en un plano a un incremento constante.

- **Fuerza:** estima la fuerza y torque a la que se encuentra sometido una parte del robot submarino.

- ***Structured light projector*:** muestra la textura en la dirección del eje Z del sistema de referencia local. Este proyector permite simular un láser y las luces del vehículo, causando sombras realistas (imagen 34).

Figura 34: Sombras generadas por el *Structures light projector*

- **Ventilación *dredge*:** elimina el barro de los objetos enterrados.

5.3. *MoveIt*

MoveIt! es un software diseñado para la manipulación móvil; incorpora los últimos avances de planificación de trayectorias, percepción 3D, cinemática, control y navegación. Además provee una plataforma para el desarrollo de aplicaciones robóticas avanzadas fácil de utilizar.

MoveIt! es una de las plataformas de fuente abierta más utilizada para la manipulación y ha sido utilizada en más de 65 robots. Su uso principal es la planificación de trayectorias para brazos robóticos, no obstante también pude ser utilizada para cualquier robot, solo hace falta definir bien sus características con este asistente.

Los robots terrestres están limitados a la navegación en 2D debido a su dinámica. No obstante, tanto los submarinos como los drones también tienen que tener en cuenta y ajustar su posición vertical, por lo que la navegación en 3D puede ser implementada. La navegación en 3D permite al AUV realizar maniobras más complejas a la hora de explorar el entorno y navegar en entornos mas complejos. Esto es especialmente útil cuando se trata de evitar obstáculos.

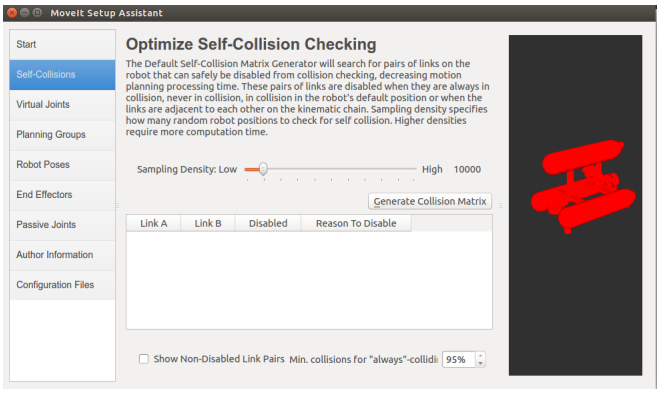


Figura 35: Interfaz MoveIt

MoveIt es un conjunto de paquetes y herramientas para realizar manipulación de robots en *ROS*. Tanto la pagina web oficial [29] como el libro [28] contienen la documentación y la lista de robots, además de ejemplos de demostración de como coger y dejar objetos y planificación de trayectorias simples.

5.3.1. Arquitectura del sistema

En esta sección se estudiará la arquitectura del asistente *MoveIt*.

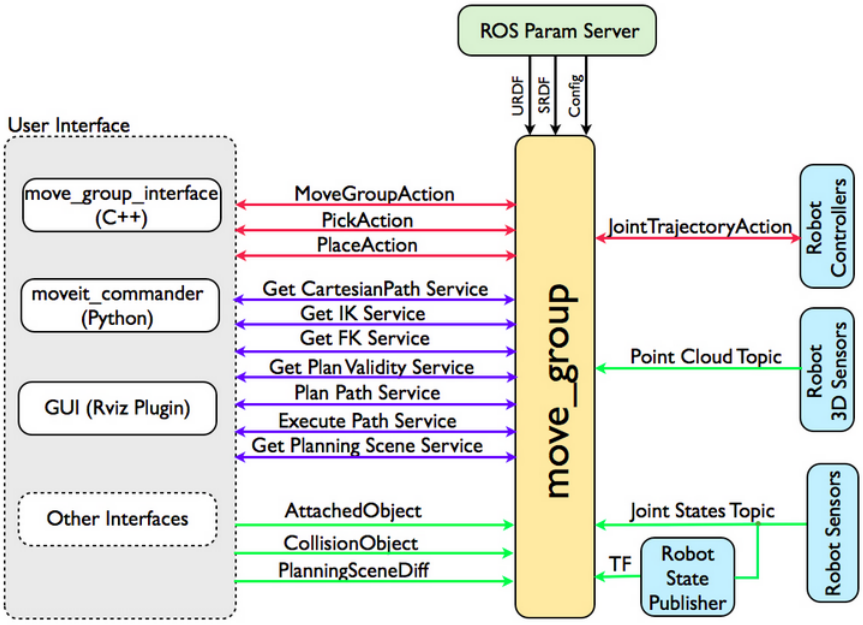


Figura 36: Diagrama de arquitectura de *MoveIt*

5.3.2. El nodo *move_group*

Como se puede observar en la figura 36, el nodo de mayor importancia es denominado *move_group*. Este nodo sirve como integrador ya que une todos los componentes individuales para dotar a *ROS* de acciones y servicios acorde a las necesidades de los usuarios.

Interfaz de usuario

Los usuarios pueden acceder a las acciones y servicios provisto del *move_group* de una de las siguientes maneras:

- **C++**: utilizando la interfaz del nodo (*move_group_interface*).
- **Python**: utiliza el paquete *moveit_commander*.
- **Mediante un GUI** (*graphical user interfaz*): utiliza el plugin *Motion Planning* de *Rviz*.

El *move_group* es un nodo de *ROS* y utiliza los parámetros del servidor de donde obtiene también los ficheros URDF y SRDF.

Este nodo utiliza los parámetros del servidor de ROS para obtener distintos tipos de información:

- URDF - *move_group* busca los parámetros de la descripción del robot (*robot_description*) para conseguir el URDF del robot.
- SRDF (Semantic Robot Description Format) - *move_group* busca los parámetros de la semántica de la descripción del robot (*robot_description_semantic*) para obtener el SRDF del robot. El SRDF se crea utilizando *MoveIt! Setup Assistant*.
- Configuración *MoveIt* - *move_group* buscará en los parámetros del servidor de *ROS* la configuración específica establecida en el asistente, como los límites de las articulaciones, la cinemática y la planificación de trayectorias entre otros. Los ficheros de la configuración se generan automáticamente y la información se almacena en el directorio de configuración correspondiente al paquete de *MoveIt* generado para el robot.

Desde el servidor de parámetros se recoge tanto la información de la cinemática del robot como la descripción del robot (*robot_description*, que es un fichero de tipo URDF), SRDF y la configuración de los ficheros. El fichero SRDF y los ficheros de configuración se generan simultáneamente que el paquete de *MoveIt* para el robot empleado. El paquete de configuración contienen los ficheros de parámetros para establecer los límites de las articulaciones, cinemática y el *end effector* entre otros.

Cuando *MoveIt* consigue toda la información del robot y su configuración, se puede decir que está correctamente configurado y por tanto se puede empezar a ordenar al robot desde la interfaz del usuario. Se puede emplear tanto el API de C++ o Python para ordenar al nodo *move_group* para realizar las acciones como coger o dejar objetos tanto en cinemática directa como inversa entre otras. Utilizando posteriormente para la planificación de trayectorias el plugin de *Rviz*, por lo que se puede ordenar al robot desde el propio *Rviz* GUI.

Como se ha discutido anteriormente, el nodo *move_group* es un integrador, por lo que no ejecuta ningún tipo de algoritmo de planificación de trayectorias. Sin embargo, conecta todas las funcionalidades como plugins. Existen plugins para los solucionadores de cinemática y planificación de trayectorias entre otros.

Después de la planificación de trayectorias, las trayectorias generadas mandan la señal a los controladores utilizando la interfaz *FollowJointTrajectoryAction*. Esto es una interfaz en la que el accionador se ejecuta en el robot y el nodo *move_node* inicia una acción que se comunica con el servidor y ejecuta la trayectoria del robot real en otro simulador. En este caso se utilizaría *UWSim*.

5.3.3. Planificación de trayectorias

Se asume que se conoce la posición inicial y la final del robot, la descripción geométrica de este y la descripción geométrica del entorno, la planificación es una técnica para encontrar el camino más óptimo en el que se mueve el robot gradualmente desde la posición inicial a la final sin tocar ningún obstáculo en el entorno y sin colisionar con los links del robot.

Tanto la descripción del robot como la del entorno, se pueden incluir dentro del mismo fichero URDF. Utilizando un sensor se puede generar el entorno en 3D, lo que puede ayudar a evitar obstáculos dinámicamente.

Dentro del URDF, en el caso de que el diseño del robot incluya mallados, es necesario incluir el fichero STL que contenga la descripción de la malla.

En el caso de haber incluido el brazo robótico del g500, el asistente debe encontrar una trayectoria en la que los links del robot nunca colisiones con el entorno ni consigo mismo, además de no violar los límites de las articulaciones.

La librera por defecto que utiliza *MoveIt* para el nodo *move_group* es la OMPL (Open Motion Planning Library). Esta biblioteca es de software libre específicamente diseñada para la planificación de movimiento, como ya se ha visto en la sección 2.5.2.

Esta biblioteca nace por la necesidad de unificación en los algoritmos desarrollados en este campo de investigación y se centra en facilitar la implementación y el uso de algoritmos basados exclusivamente en muestreo aleatorio. Puesto que la biblioteca solo esta diseñada para la planificación de movimiento de forma lo más general posible, no incluye detectores de colisiones ni se limita a la representaciones del espacio de forma específica, así como tampoco incluye ningún medio de visualización de los resultados. Esto es así para no hacer necesario utilizar la OMPL con un determinado detector de colisiones o una interfaz de usuario. El objetivo es que el usuario pueda utilizar el contenido de esta biblioteca en otros sistemas que ya posean el resto de componentes necesarios para cada caso particular.

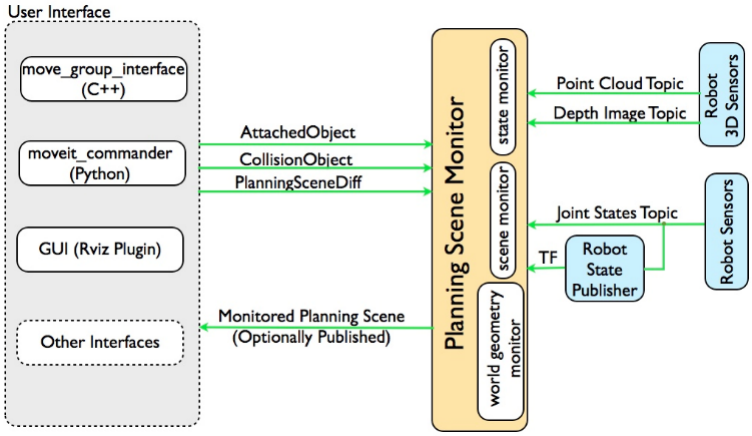


Figura 37: Planificador de escenas

La librería OMPL incluye la implementación de numerosos algoritmos como el PRM, RRT, EST, SBL, KPECE y SyCLOP entre otros.

Se pueden añadir más restricciones para los planificadores de movimiento como de posición, que limita la posición de un link, de orientación, de visibilidad, de articulación o definidas por el usuario.

Utilizando estas restricciones, se puede enviar la solicitud de la trayectoria y el planificador generará una trayectoria válida acorde a estas restricciones.

5.3.4. Planificación del entorno

El término planificación del entorno se utiliza para representar el entorno al rededor del robot y además almacena la información el robot en si. La planificación de escenas se monitoriza dentro del nodo *move_group* que mantiene la representación de la escena. El nodo *move_group* contiene otra sección denominada **world geometry monitor**, que construye la geometría del mundo a partir de los sensores del robot y de la participación del usuario.

MoveIt tiene soporte para dar soporte a dos tipos de *inputs*:

- **Nubes de puntos**: manejado por el plugin *point cloud occupancy map updater*.
- **Imágenes**: manejadas por el plugin *depth imagen occupancy map updater*.

5.3.5. Comprobación de colisiones

La comprobación de colisiones se configura dentro de una planificación de la escena utilizando el objeto *Collision World*. Afortunadamente, MoveIt realiza la organización para que el usuario nunca tenga que preocuparse de que exista una colisión. La comprobación de la colisión se realiza utilizando el paquete FCL, que es la principal librería de *MoveIt*.

MoveIt permite realizar comprobaciones de colisiones para distintos tipos de objetos, incluyendo mallas, formas primitivas o el octomap.

La comprobación de colisiones es una operación de elevado coste, normalmente está cerca del 90 % del coste de la planificación de trayectorias. La ACM (*Allowed Collision Matrix*) incluye un valor binario que se corresponde con la necesidad de comprobar la colisión entre pares de objetos (tanto si forman parte del robot como si son del entorno). Si el valor entre dos objetos es 1, indica que no es necesario realizar una comprobación de colisión.

5.3.6. Procesamiento de trayectorias

Los planificadores de trayectorias normalmente solo generan trayectorias, lo que quiere decir que generalmente no hay tiempos asociados a estas. *MoveIt* incluye un fichero con información relativa a las velocidades y aceleraciones máximas impuestas en las articulaciones, por lo que calcula el tiempo necesario para que el robot realice la trayectoria. Esta información se almacena en el fichero *joint_limits.yaml*.

5.4. Rviz

Rviz es un entorno de visualización en 3D en *ROS* (figura 38), que permite la observación de los datos de los sensores y la información del estado de *ROS*. Utilizando *Rviz* se puede visualizar la configuración actual de un modelo virtual de robot, se pueden mostrar representaciones en vivo de los valores de los sensores que se generan sobre los *topics* de *ROS*, datos de la cámara, etc. Esta herramienta se emplea a lo largo del proyecto y con ella se observará el funcionamiento de las aplicaciones implementadas a lo largo del mismo.

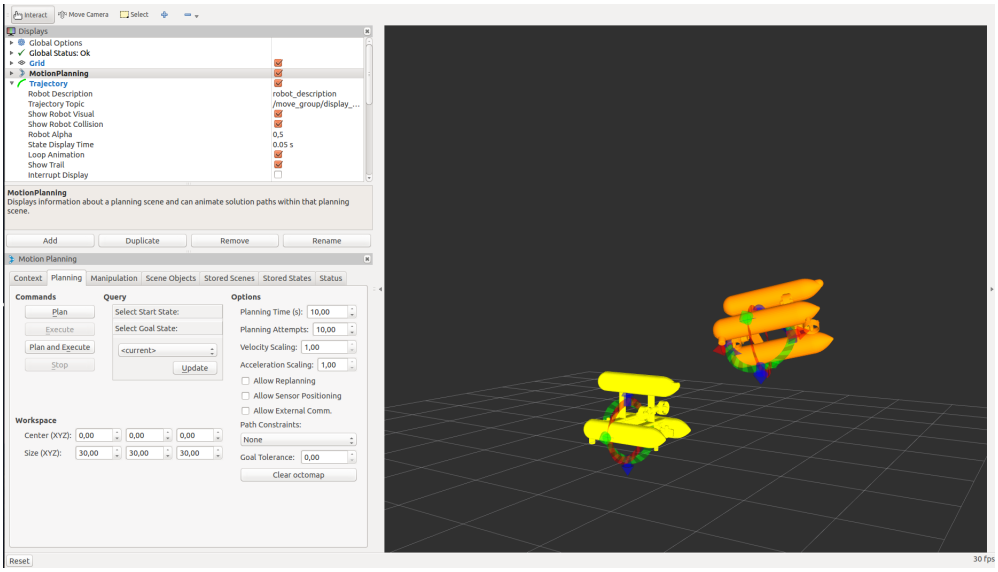


Figura 38: Ventana *Rviz*. El submarino amarillo representa la posición inicial, mientras que el naranja representa la posición final

Principalmente hay dos maneras de introducir información en el entorno de *Rviz*:

- **Información proveniente de sensores.** *Rviz* comprende información proveniente tanto de escáneres láser, nubes de puntos, cámaras o ejes de coordenadas y tiene métodos específicos de mostrar los mismos, en función de las preferencias del usuario.

- **Marcadores de visualización.** Permiten al programador enviar la información en forma de cubos, flechas, etc.

La combinación de la información obtenida a través de la combinación de los sensores y los marcadores personalizados es lo que hace de *Rviz* una herramienta muy potente cuando se trata de desarrollar las capacidades de un robot. Esta combinación es lo que se denomina el *stack* de navegación de *ROS* y muestra información de obstáculos, una rejilla vóxel 3D y un mapa topológico. Los marcadores de *Rviz* actualmente se emplean en proyectos que implican detección de objetos o calibración.

Por último, se puede observar en la imagen 39 como se comunican todos los nodos cuando el fichero al completo es ejecutado y se muestra como *Rviz* se ha subscrito a todos y cada uno de los *topics* que publican los paquetes que se han descrito..

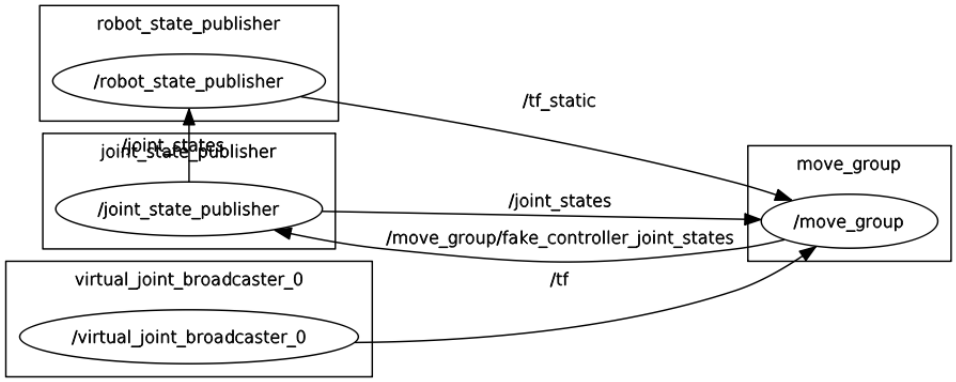


Figura 39: Interacción entre nodos

5.5. *Gazebo*

Gazebo es un simulador de software libre de robots, con el que se pueden diseñar robots, realizar simulaciones utilizando escenarios realistas tanto interiores como exteriores.

El simulador cuenta con acceso a múltiples motores de físicas que determinan como interactúan objetos entre sí al colisionar. También cuenta con gráficos 3D avanzados, que proveen renderizados realistas de entornos, incluyendo luces, sombras y texturas.

Es un simulador completo en lo que a sensores se refiere, capaz de recopilar datos de cámaras, sensores de contacto, etc. gracias a lo cual se pueden realizar simulaciones muy precisas del comportamiento del robot. Entre los robots disponibles se encuentran PR2, Pioneer DX, REEM o NAO, pero además el simulador contiene una herramienta para diseñar un robot propio, por lo que en la práctica este simulador es válido para cualquier robot.

Por el hecho de ser un software libre, *Gazebo* cuenta con multitud de plugins externos que hacen de este software un simulador más completo y capaz de adaptarse a las necesidades de cada situación.

Aunque sea un simulador muy completo no cuenta con la posibilidad de incorporar entornos acuáticos. Es por ello que no se ha utilizado en la resolución del problema planteado.

Otras herramientas

Para poder instalar herramientas necesarias para este proyecto que carecen de un fichero ejecutable en *Ubuntu*, como son *Solid Edge* y *SketchUp*, se ha instalado *Wine*. A continuación se realiza una breve explicación de cada una de ellas.

5.6. *Wine*

Wine (*Wine Is Not an Emulator*) es una reimplementación de la interfaz de programación de aplicaciones de Win16 y Win32 para sistemas operativos basados en Unix. Permite la ejecución de programas diseñados para otros sistemas operativos, en este caso para *Windows*.

Wine provee de un conjunto de herramientas de desarrollo para portar código fuente de aplicaciones *Windows* a *Unix*. Además es un cargador de programas, el cual permite que muchas aplicaciones para *Windows* se ejecuten sin modificarse en varios sistemas operativos *unix*.

5.7. *Solid Edge*

Solid Edge [43] es un programa parametrizado de diseño asistido por computadora de piezas tridimensionales. Permite el modelado de piezas de distintos materiales, doblado de chapas y ensamblaje de conjuntos entre otros.

Con este programa se ha diseñado la jaula en la que se encontrará el robot en la simulación. El objetivo es que salga de esta sin colisionar con ningún barrote y el g500 sea capaz de alcanzar su objetivo.

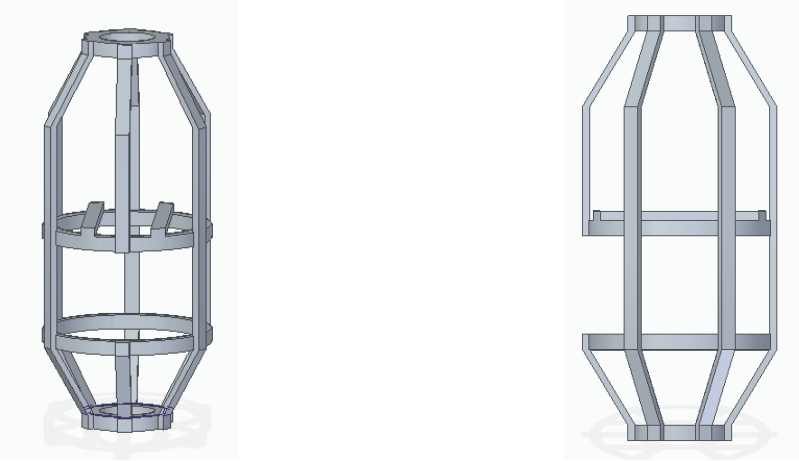


Figura 40: Jaula del robot

5.8. *SketchUp*

SketchUp [69] es un software de modelado en 3D basado en caras.

En la elaboración del entorno este programa ha sido utilizado para diseñar la mina en la que se encuentra el submarino en la simulación.

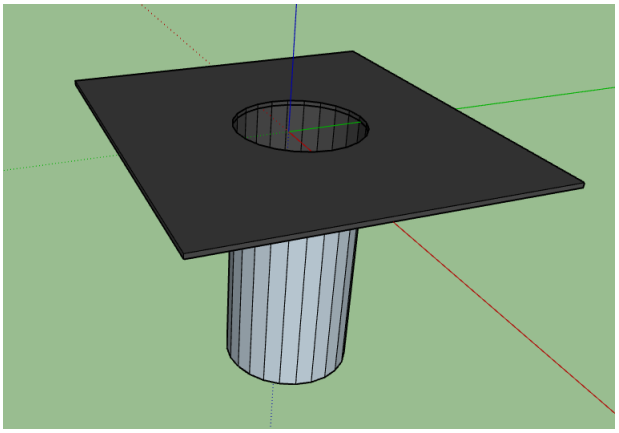


Figura 41: Mina

5.9. *Blender*

Blender es un programa de diseño gráfico utilizado para realizar diseños 3D y animaciones entre otras. En este proyecto, no obstante se ha utilizado simplemente para transformar archivos del formato STL o DAE a 3DS.

Figura 42: Logo Blender

6. Marco regulador

6.1. Legislación

Los avances científicos y tecnológicos en la ingeniería marina, civil y militar, han permitido el desarrollo y auge de la industria de los denominados drones submarinos o AUVs, unos vehículos capaces de operar bajo el agua y en una superficie.

A diferencia de los drones aéreos, los drones marinos y submarinos no cuentan con una regulación propia y han pasado más desapercibidos en los medios de comunicación; ya que las aeronaves civiles pilotadas por control remoto han sido objeto de una regulación específica, ocupando así numerosos titulares en los medios.

Los robots marinos cada vez tienen un uso más extendido en tareas como la cartografía, el peritaje, la vigilancia y salvamento marítimo, la prevención y lucha contra la contaminación marina, la seguridad y defensa y otras actividades oceanográficas.

Resulta muy complicado establecer un marco jurídico unitario para unos vehículos que difieren entre sí en características, diseños, dimensiones y uso.

Desde una perspectiva legal, y ante la variedad de características que pueden presentar estos vehículos, no sorprende que el estudio jurídico de estos vehículos no esté claramente definido. Sujeto a la actual Ley 14/2014, de 24 de julio, de Navegación Marítima, muchos de estos submarinos no tienen un encaje cómodo en la definición de buque (ya que requeriría poder transportar personas o cosas); ni el concepto de artefacto naval (requeriría quedarse situado en un punto fijo).

Sin embargo, a pesar de no estar regulados, estos vehículos operan en el mar y por ello están expuestos a riesgos derivados de la actividad marítima (sean o no relacionados con la actividad marítima), ya que pueden ser objeto de relaciones jurídicas propias de los buques y artefactos navales, tales como los contratos de construcción o el régimen de la responsabilidad civil derivada de daños causados por contaminación marítima.

Por todo ello, se debe plantear la necesidad de regular expresamente el uso de los submarinos, especialmente ante la previsión de crecimiento de esta industria.

Desde le punto de vista de los riesgos, y a pesar de que la Ley de Navegación Marítima no incluye expresamente a este tipo de vehículos dentro de los intereses asegurables objeto del seguro marítimo, la propia ley prevé que pueden ser objeto de seguro marítimo cualesquiera intereses patrimoniales legítimos expuestos a los riesgos de la navegación marítima, lo que incluiría a los AUV, o al menos, a aquellos AUVs o ROVs [76].

7. Trabajo Experimental

El presente apartado tiene como objetivo describir como se ha conseguido realizar la trayectoria para que el AUV (GIRONA 500) alcance la posición final conociendo la posición de la que parte.

El proceso que posibilita alcanzar el objetivo final del trabajo que concierne a esta memoria se compone de distintas etapas. En la figura 43 se puede observar el proceso llevado a cabo para la resolución del problema.

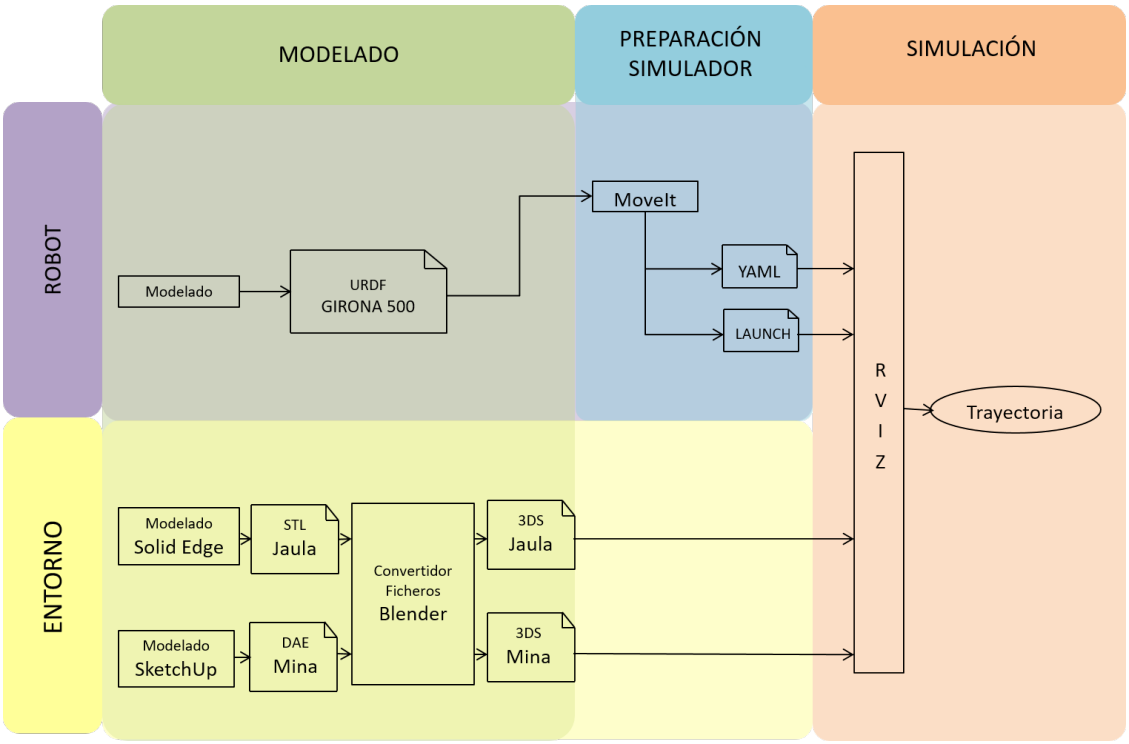


Figura 43: Arquitectura del proyecto

Debe diferenciarse la realización del proyecto en 3 etapas:

- **Modelado:** incluye las descripciones tanto del robot como del entorno.
  - **Robot:** cómo se ha obtenido la descripción del submarino.
  - **Entorno:** diseño del entorno en el que se encuentra el AUV, para que la simulación sea lo más verosímil posible.

- **Preparación de la simulación:**
  - **Robot:** se ejecutará el asistente *MoveIt* para generar los archivos necesarios.

- **Simulación:** cómo se han generado las trayectorias con la herramienta *Rviz*.

No obstante, para una mayor simplificación a la hora de describir el proceso se diferenciará en tres bloques:

- **Robot:** se incluirá la descripción del proceso hasta el punto anterior a abrir el generador de trayectorias *Rviz*.

- **Entorno:** descripción de las herramientas y que se ha diseñado con cada una de ellas.

- **Simulación:** una vez que se tienen todas las descripciones, pasos seguidos para la generación de trayectorias.

Como se mencionó anteriormente, este proyecto está realizado con el submarino GIRONA 500, ya que no se dispone de la descripción del robot que la Universidad Carlos III está construyendo. No obstante, el proceso realizado podría realizarse con cualquier otra descripción de un submarino.

7.1. Robot

En primer lugar es necesaria la descripción del robot para lo cual se instaló el UWSim explicado en el anexo A.3. La descripción del robot se encuentra bajo la carpeta donde se haya instalado el simulador, en este caso esta bajo la dirección */opt/ros/indigo/share/uwsim/data/scenes* como se puede observar en la imagen 44.

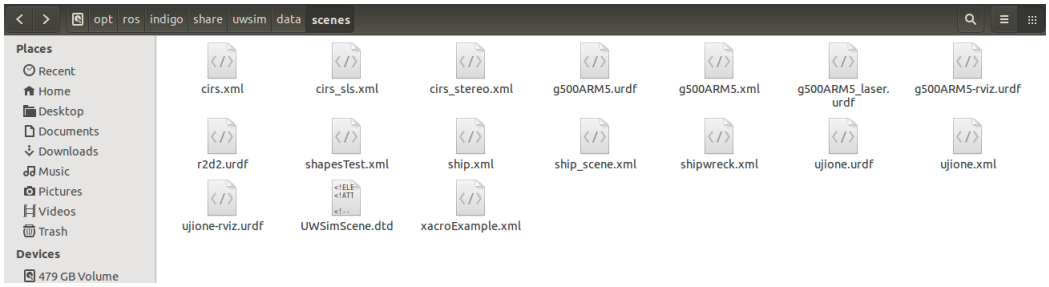


Figura 44: Carpeta donde se encuentra la descripción del simulador *UWSim*

Dentro de esta carpeta existen numerosas descripciones, todas ellas relacionadas con la descripción del entorno y del robot utilizado. Además se puede observar dos tipos de ficheros: los XML y los URDFs.

- **XML *eXtensible Markup Language*:** el XML es un lenguaje que se emplea para representar información estructurada, de modo que esta información pueda ser almacenada, transmitida, procesada, visualizada e impresa por diversos tipos de aplicaciones y dispositivos.

  XML da soporte a bases de datos, siendo de gran utilidad cuando varias aplicaciones deben comunicarse entre sí o integrar información, como se da en esta situación [70].

- **URDF:** un URDF es un paquete que contiene numerosos ficheros XML con descripciones de robots, sensores de estos y escenas en las que se desarrollará la simulación entre otras. Cada XML tiene un analizador sintáctico (conocido en inglés como *parser*) para uno o más lenguajes.

A continuación, en la imagen 45, se puede apreciar y comprender la relación entre los componentes que forman el URDF.

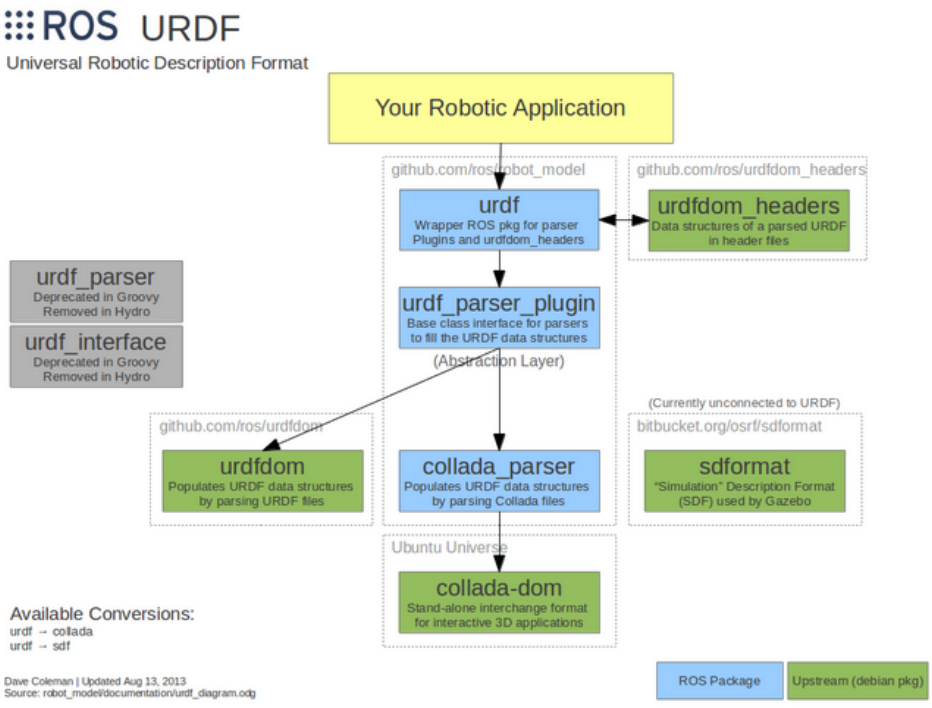


Figura 45: Estructura del URDF

Algunos de las estructuras de datos mostrados en la imagen 45, como *urdfdom* y *urdfdom_headers* son completamente independientes de *ROS*.

Existe una diferencia entre un fichero URDF y una estructura de datos URDF; el primero sigue la estructura del XML, mientras que la estructura de datos URDF es un conjunto de clases genéricas procedentes de varios formatos, como son URDF y Collada [71].

Por tanto los ficheros XML describen el escenario o la escena donde se va a desarrollar la simulación y los parámetros correspondientes; mientras que los robots son descritos en ficheros URDF. En el caso de este simulador, al igual que en la mayoría de ellos, el fichero XML contiene una referencia al fichero URDF para incluir al robot en la escena.

Entre los ficheros que se muestran en la carpeta, solo algunos son relevantes a la hora de desarrollar este proyecto. Entre ellos destacan [46]:

- **g500ARM5.xml:** este fichero contiene un modelo 3D del submarino GIRONA 500 con un brazo manipulador denominado ARM5.

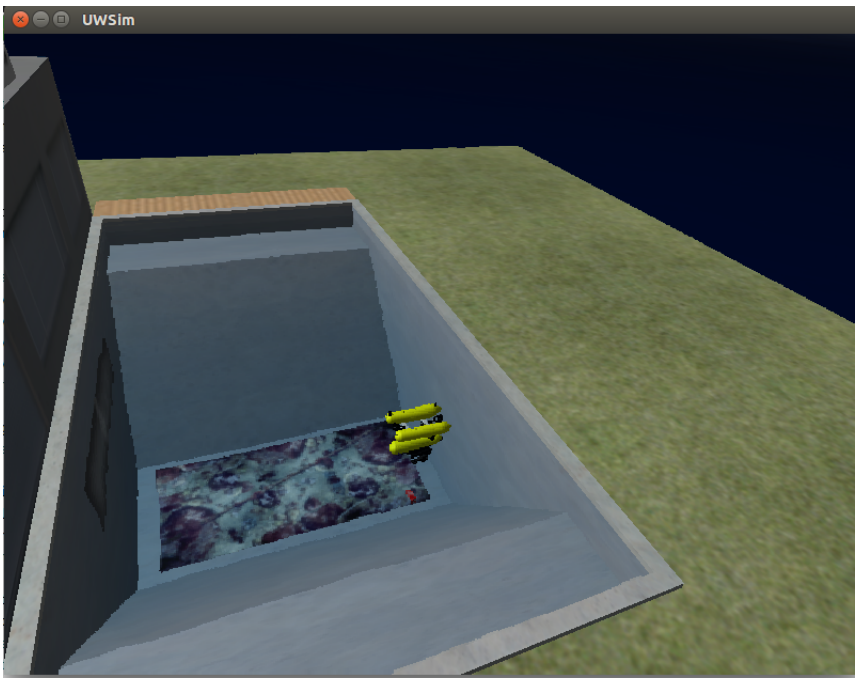


Figura 46: Entorno del simulador *UWSim*

- **cirs.xml:** en este fichero se incluye una reproducción de las instalaciones CIRS (se encuentran en la Universidad de Gerona). Contiene un tanque de agua de 8x16 metros con una profundidad máxima de 5 metros, como se puede observar en la imagen 46. Este escenario contiene al g500 con la descripción del ARM5 además de las dos cámaras que posee a bordo, el sensor de rango y todos los demás sensores analizados en la sección 5.2.3.3.

- **cirs_stereo.xml:** contiene la misma descripción del entorno 3D (la descripción contenida en el cirs.xml); la única diferencia es la cámara del submarino, ya que en este caso tiene una cámara estéreo en vez de dos cámaras normales.

7.1.1. URDF GIRONA 500

Una vez que se conoce el contenido de los ficheros se puede proceder a analizar el AUV g500.

Existen comandos para la terminal que permiten comprobar si la descripción del robot en el URDF es correcta y otros que generan un diagrama en el que se muestra la descripción del URDF.

En primer lugar, comprobar que el URDF está bien generado es de gran utilidad, ya que no ser correcto ningún simulador o asistente podría abrirlo. En caso de que hubiese errores en la terminal aparecería la primera línea en la que se encuentra el error y así sucesivamente hasta que el código estuviese correctamente formulado. El comando que realiza esta comprobación es el siguiente:

```
check_urdf g500.urdf
```

El diagrama que contiene la información del URDF se obtiene con el comando:

```
urdf_to_graphiz g500.urdf
```

La descripción del robot incluye un brazo robótico para manipular objetos como se explicó en la sección 2.1.5; no obstante el brazo no es de utilidad para resolver el problema planteado, por lo que se ha modificado el URDF inicial eliminando la descripción del ARM5, ya que como se ha mencionado anteriormente, el objetivo del proyecto es la navegación de un robot y no la manipulación con brazos robóticos.

Al modificar el URDF y reducirlo únicamente a las partes que afectaban a la resolución del problema, éste quedaba simplificado a la figura 47, donde se puede observar que solo se dispone del *base_link* ya que es el único link empleado en la descripción.

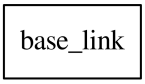


Figura 47: Estructura del GIRONA 500 sin el ARM5

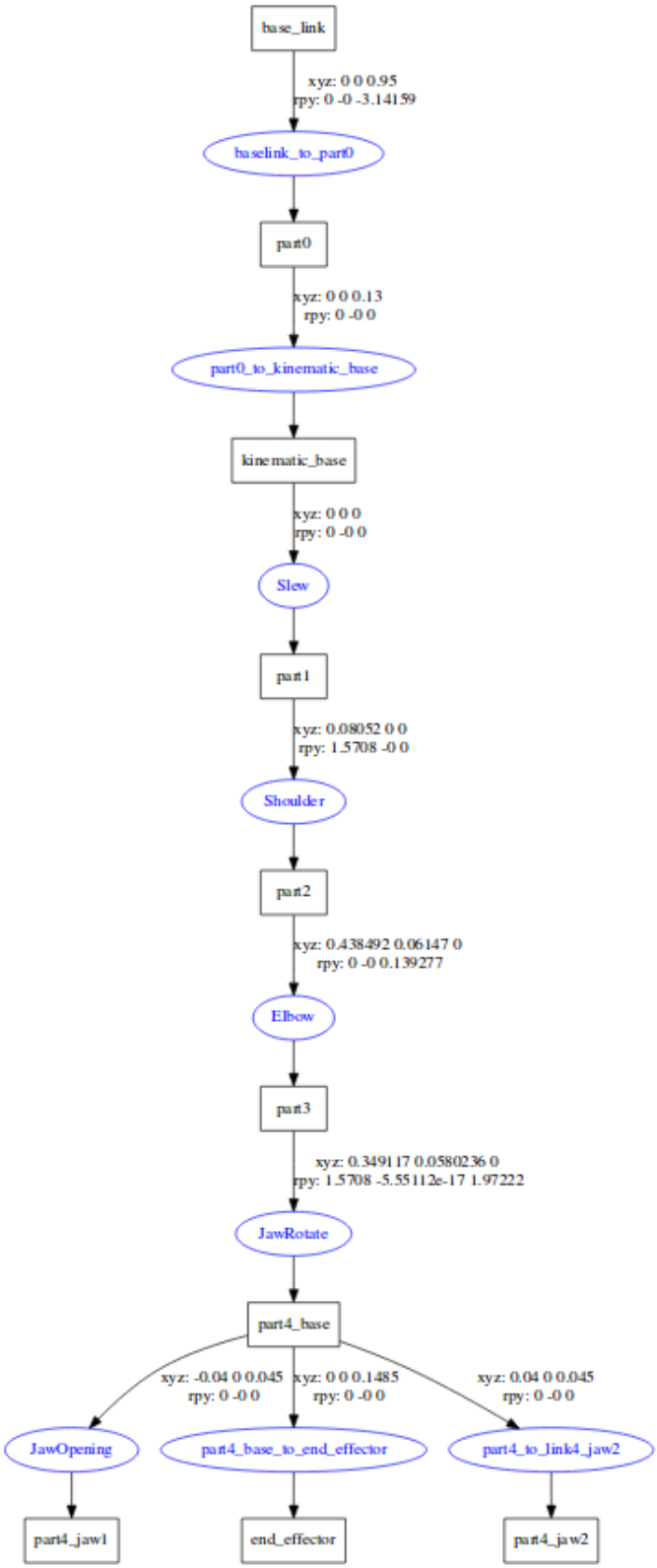


Figura 48: Estructura del GIRONA 500 con el ARM5

Sin embargo, en caso de haber empleado la descripción completa del robot se habría obtenido el siguiente diagrama (imagen 48), mucho más completo que el anterior.

En este caso, como se puede observar, el URDF contiene más de un link (representados en los rectángulos), además de contener la descripción de las articulaciones entre cada uno de los links del sistema (representado en los óvalos). El *base_link* representa la estructura del submarino en sí, mientras que el resto de links son los que definen el brazo robótico.

Los URDFs pueden incluir la descripción del robot de dos maneras diferentes.

- **Formas geométricas simples.** Estos archivos son más sencillos, ya que se forman a partir de figuras geométricas simples como sería un cilindro, un cubo o una esfera entre otras.

- **Con mallado.** En el caso de querer incluir descripciones de formas más complejas se utilizará la descripción con mallado. Además, se utiliza cuando se desean comprobar las colisiones entre distintos grupos que conforman el robot.

  Existen distintos formatos para definir este mallado, ya que en función del simulador o la herramienta que se vaya a emplear se utilizará una u otra.

  - **OSG (*OpenSceneGraph*):** es un motor gráfico de código abierto utilizado para el desarrollo de aplicaciones como simulación visual y modelado 3D.

    Este formato es el que emplea *UWSim* en su simulación.

  - **STL:** es un formato de archivo informático de diseño que define la geometría de objetos 3D. No incluye información sobre color, texturas o propiedades físicas.

    Este formato de mallado es el que el asistente *MoveIt* puede interpretar.

  - **COLLADA (*COLLAborative Design Activity*):** es un formato con la estructura de un XML. La extensión de estos archivos es .dae (*digital assets exchange*). Este formato de archivo de intercambio 3D se utiliza para el intercambio de archivos digitales entre varios programas de gráficos [75].

  - **IVE:** esta extensión también representa un modelo 3D creado con OpenSceneGraph (OSG)

7.1.2. Preparación del robot

Una vez que se dispone del URDF correcto se procede a la elaboración de los ficheros necesarios para poder visualizar el g500 con el simulador *Rviz*. Para generar estos ficheros, se cuenta con la ayuda del asistente *MoveIt* (explicado en la sección 5.3), ya que permite que se creen todos los archivos necesarios para la correcta visualización del submarino.

Al asistente hay que proporcionarle el URDF además de definir algunos aspectos como la cadena cinemática deseada para el control. El resto de información sobre la cadena, restricciones, masas y direcciones del eje se definen a su vez de forma automática por la información proporcionada por el URDF.

Cuando se completa el asistente, al generar los archivos necesarios, lo que realmente se está llevando a cabo es la parametrización del nodo *move_group* ya que es el núcleo que proporciona la funcionalidad del software.

Una vez cargado el URDF en el asistente se irán siguiendo los pasos establecidos por el asistente:

- **Autocomprobación de colisiones *Self-Collision Checking*:** el generador de las colisiones entre los pares de links del robot, puede desactivar que se compruebe la colisión entre ellos, lo que reducirá el tiempo de generación de las trayectorias.

  En este caso, como solo se dispone de un link, no hace falta generar la matriz de colisiones.

- **Articulaciones virtuales *Virtual joints*:** define una articulación virtual entre un link del robot y un marco de referencia externo a este.

  Para la resolución del problema es necesario crear una *virtual joint* entre el submarino en sí y el entorno. El tipo de relación entre ellos será flotante, pues el submarino es un robot con múltiples grados de libertad.

- **Planning groups:** sirve para describir distintas partes del g500, como en el caso de haber utilizado la descripción completa del robot (con el brazo extensible) se definiría el end effector.

- **Posiciones del robot:** permite añadir posiciones predefinidas.

- **End effectors:** permite configurar los agarres y end effectors del robot.

- **Articulaciones pasivas *Passive joints*:** permite al usuario definir cualquier articulación que no se desee planificar cinemáticamente.

Una vez completados todos los pasos, se genera un paquete (*g500_moveit*) que contiene los ficheros relacionados con la configuración del AUV y los *launch* para el *move_group*. Por tanto, se procede al desarrollo del entorno donde la simulación tendrá lugar.

7.2. Entorno

El entorno en el que se plantea la simulación, es el mismo que se pretende crear cuando la universidad construya el submarino. El submarino se encontrará dentro de una jaula que a su vez estará situada en el interior de una mina, donde el submarino llevará a cabo distintas misiones.

Se han creado ambos archivos con programas de diseño en 3D como son *Solid Edge* (descrito en la sección 5.7) y *SketchUp* (descrito en la sección 5.8. Los archivos que se han obtenido con estos programas no son compatibles con la herramienta que se utilizará posteriormente para la generación de trayectorias.

El fichero de la jaula generado es del tipo STL, mientras que el de la mina es del tipo DAE. El simulador no permite visualizar ninguno de estos archivos, por lo que se ha utilizado un tercer programa, *Blender* (sección 5.9) para transformar estos archivos al formato 3DS.

7.3. Simulación

Como ya se ha explicado anteriormente en este documento, los simuladores son una parte muy importante de la robótica, y antes de realizar cualquier prueba sobre el robot real, es necesario ver si los programas desarrollados funcionan de manera satisfactoria sobre una simulación.

La generación de trayectorias se ha llevado a cabo con el paquete *Rviz*, por lo que es necesario en primer lugar lanzar el simulador. Para ello, hay que escribir en la terminal el siguiente comando, una vez que se está dentro del *workspace* en el que se encuentra el paquete:

```
roslaunch g500_moveit demo.launch
```

Rviz comenzará analizando si el archivo no tiene ningún error en la compilación, en caso de tener alguno se indicaría en la terminal el tipo de éste. Si por el contrario los ficheros son correctos, el programa se abrirá apareciendo la simulación

del submarino a la derecha, mientras que a la izquierda aparecerá un menú con todas las opciones que ofrece este visualizador (ver imagen 49) . Además, dentro de la pestaña de *motion planning*, se puede observar la librería de algoritmos que se empleará en la resolución de las trayectorias (imagen 50).

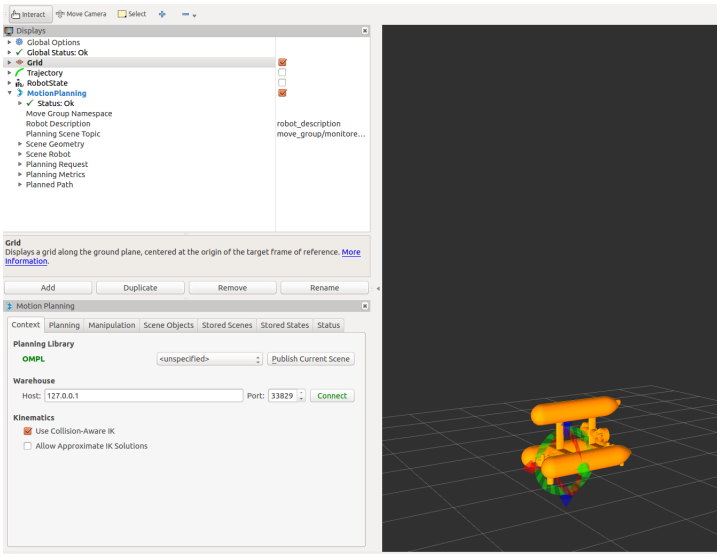


Figura 49: *Rviz*

A la hora de planificar trayectorias con *Rviz* se puede elegir el algoritmo que se desea emplear, en caso de seleccionar *unspecified* se utilizará por defecto LBKPIE-CE en el caso de tener un espacio de estado con una proyección determinada. En caso de no tenerla se emplearían por defecto los algoritmos RRTConnect y RRT.

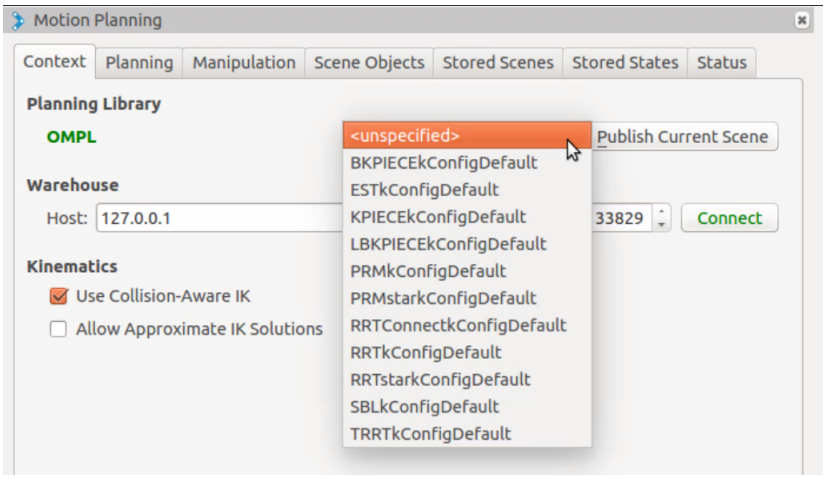


Figura 50: Algoritmos existentes en la librería OMPL

Como se puede observar en la imagen 50 dentro de *Motion Planning* existen numerosas pestañas, que ayudan a configurar las posiciones del robot, y los obstáculos presentes en el entorno.

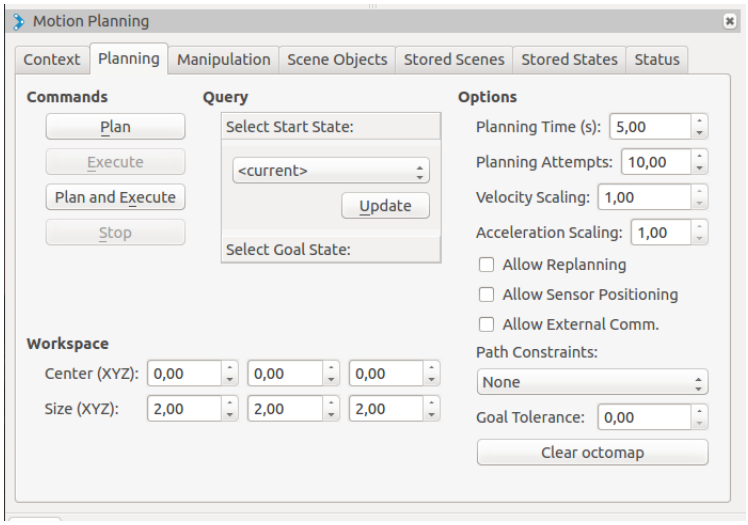


Figura 51: *Rviz* pestaña de planificación de trayectorias

En la pestaña que se muestra en la imagen 51 se introduce la información relacionada con la trayectoria, como son los puntos inicial y final, el espacio dentro del que puede trabajar el robot (*workspace*) y el tiempo del que dispone el robot para alcanzar la meta entre otros.

Para facilitar la visualización de la posición inicial del AUV se representa en amarillo, la posición final en naranja y los objetos del entorno en verde. En caso de que existiesen colisiones en la simulación aparecería en rojo.

Otra de las pestañas existentes dentro de *Motion Planning* es *Scene Objects* donde se introducen los objetos presentes en la simulación. Además se puede rotar los obstáculos y posicionarlos en el lugar adecuado. En este caso se introducirá la jaula en la que se encuentra el robot en la posición inicial (ver imagen 52) y la mina donde se llevará a cabo la misión.

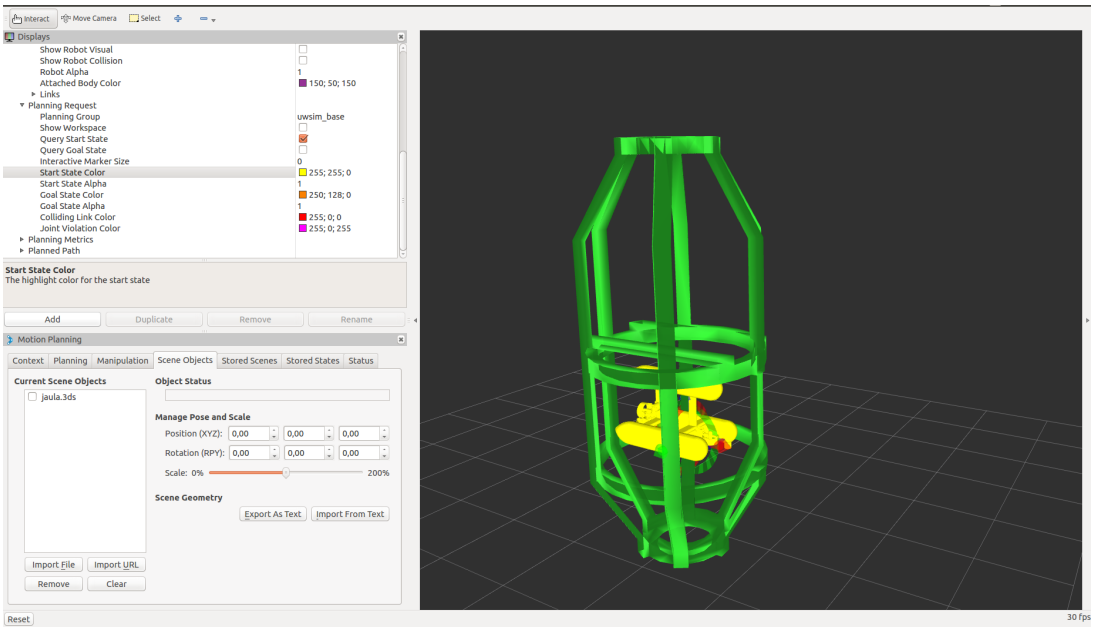


Figura 52: Configuración del entorno *Rviz*

Una vez seleccionadas e introducidas todas las características que se desean simular, se procede a la generación de la trayectoria esquivando obstáculos. Para ello, en la pestaña *Planning* se seleccionará plan (ver imagen 53).

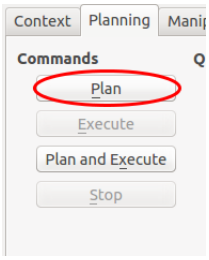


Figura 53: Generación de trayectorias

En el caso de que el robot se encuentre en una posición compleja o los puntos inicial y final se encuentren muy distanciados, es posible que tarde más tiempo en encontrar una trayectoria que permita alcanzar el objetivo. En estos casos, se debe aumentar el tiempo de planificación (*planning time*).

8. Resultados

El resultado de este proyecto es la trayectoria en sí, como se puede observar en la figura 51.

Se han generado numerosas trayectorias con algunos de los distintos algoritmos existentes en el visualizador *Rviz*. A continuación se muestran algunas.

Aunque la posición inicial y final del submarino sea la misma en todos los casos, los tiempos de planificación de la trayectoria son distintos, ya que en función del método es más sencillo encontrar una trayectoria.

La primera simulación realizada (imagen 54) se realiza sin especificar el algoritmo con el que se desea calcular la trayectoria, por lo que por defecto se emplea el algoritmo LKBPIECE. El tiempo necesario para calcular la trayectoria varía de unas ejecuciones a otras, siendo en torno a 1 segundo. En este caso, no obstante el tiempo de planificación es de 0.771 segundos.

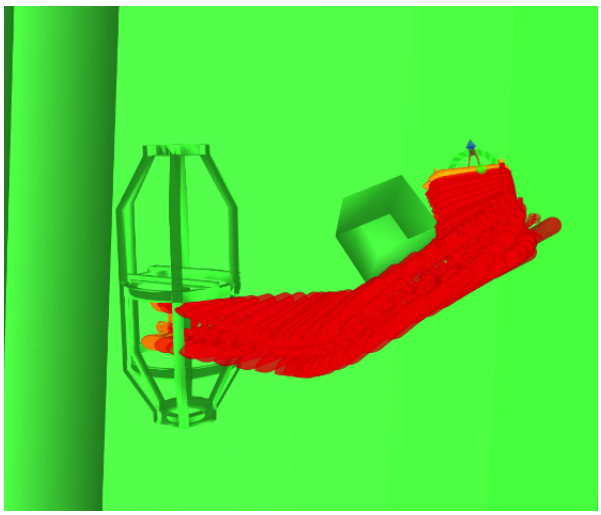


Figura 54: Algoritmo LKBPIECE

Para generar la segunda simulación se ha establecido que se planifique la trayectoria acorde al algoritmo BKPIECE (imagen 55). El tiempo necesario para la planificación de esta trayectoria es de 1.874 segundos.

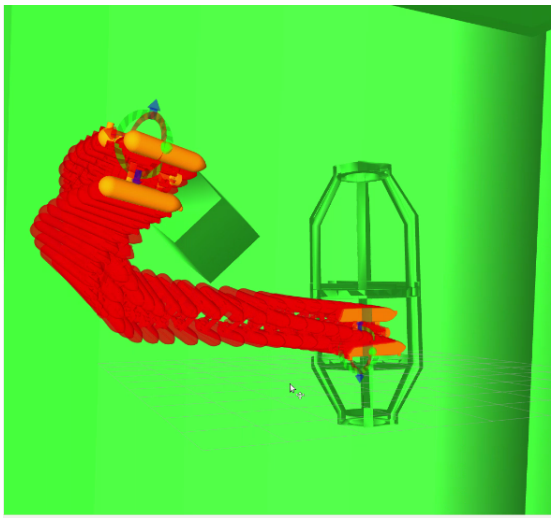


Figura 55: Algoritmo BKPIECE

Como se puede observar en esta segunda simulación el submarino pasa más alejado del obstáculo que en el caso anterior.

En la tercera planificación de trayectorias se ha empleado el algoritmo RRTk-Connect (imagen 56). En este caso se han realizado dos simulaciones (sin variar nada del entorno ni las respectivas posiciones del g500) para observar las diferencias dentro del mismo algoritmo.

En la primera simulación con este algoritmo (imagen 56), el tiempo empleado ha sido de 3.154 segundos, mientras que en la segunda (imagen 57) el tiempo necesario para la planificación es mucho mayor (20.001 segundos).

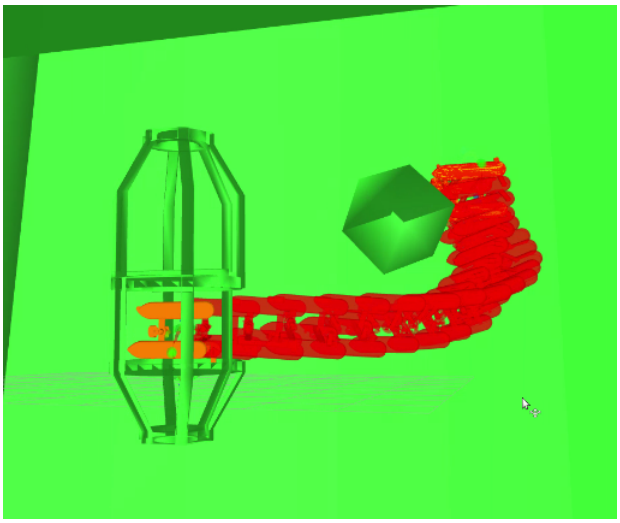


Figura 56: Algoritmo RRTkConnect

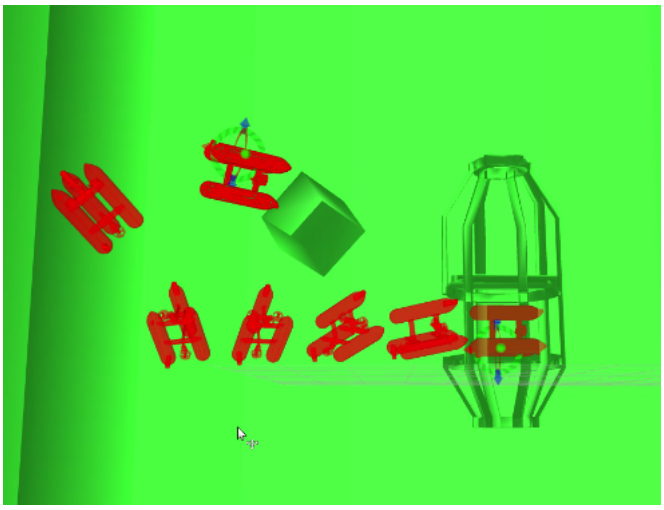


Figura 57: Algoritmo RRTkConnect

Como se puede observar en estas imágenes en la primera ejecución con este algoritmo el submarino continua en línea recta hasta el último segundo que tiene que girar para situarse en la posición establecida como posición final; mientras que en la segunda ejecución el robot se gira primero y después continua marcha atrás.

La siguiente planificación de trayectorias se ha realizado con el algoritmo RRT-Connect (imagen 58) y el tiempo empleado en la generación de la trayectoria es de 1.516 segundos.

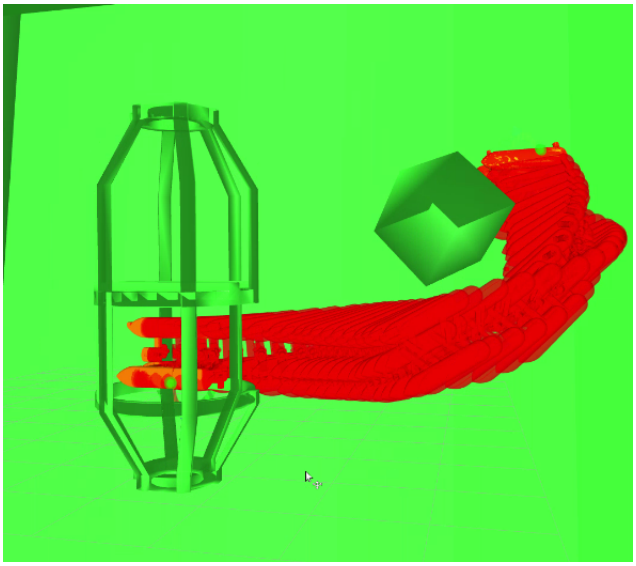


Figura 58: Algoritmo RRTConnect

Por último, en esta simulación se ha empleado el algoritmo RRTStar (imagen

59) y el tiempo de planificación ha sido 20.004 segundos.

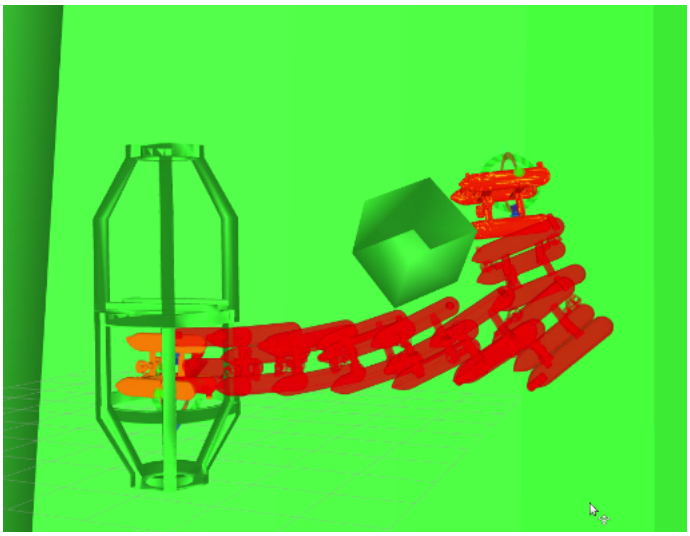


Figura 59: Algoritmo RRTStar

En la tabla 9 se recogen los tiempos empleados en la generación de trayectorias.

Algoritmo	Tiempo (s)
LBKPIECE	0.771
BKPIECE	1.874
RRTkConnect	3.154
RRTConnect	1.516
RRTStar	20.004

Tabla 9: Tiempo de planificación para cada algoritmo

Por tanto, la trayectoria generada puede variar incluso empleando el mismo algoritmo y cada algoritmo necesita un número distinto de estados para poder generar correctamente la trayectoria.

9. Conclusiones y futuros trabajos

9.1. Conclusiones

Una vez se ha realizado el proyecto completo, en este capítulo se recogerán todas las conclusiones obtenidas a lo largo del mismo.

La planificación y generación de trayectorias es una herramienta clave para la navegación autónoma de robots móviles debido a la necesidad de poseer una trayectoria óptima libre de obstáculos.

A pesar de que existen múltiples algoritmos empleados en la planificación y generación de trayectorias, no existe uno que sea superior al resto en todos los aspectos; sino que cada uno tiene sus fortalezas y debilidades. Es por ello que hay que tener en consideración el problema a resolver y las características y limitaciones propias de cada caso concreto, pudiendo obtener así la solución más óptima.

De los distintos métodos para la planificación de trayectorias estudiados, se podría concluir que los más óptimos para un entorno en 3D (como el que se dispone en esta situación, en la que el robot se encuentra en una mina) son los métodos basados en muestreo aleatorio. Este tipo de método presenta un coste computacional menor que los algoritmos combinatorios para espacios grandes con un elevado número de dimensiones. Los métodos basados en control también son poco costosos computacionalmente, sin embargo, en entornos complejos no se obtienen las soluciones óptimas.

El algoritmo que se utiliza por defecto en la librería OMPL en caso de tener un espacio de estado con una proyección predeterminada es LBKPIECE, ya que se ha demostrado que este planificador funciona de manera consistente en multitud de problemas de planificación de movimiento en el mundo real. Por lo tanto. Por el contrario, si el espacio de estado no tiene una proyección determinada se emplearán los algoritmos RRTConnect o RRT (en función de si se puede emplear un planificador bidireccional o no).

En el ejemplo resuelto con el submarino g500, al disponer de un espacio de estado con una proyección predeterminada (ya que se ha empleado uno de los espacios de estado incorporados), se ha empleado el algoritmo LBKPIECE. Este método utiliza dos árboles de exploración, uno en el estado inicial y otro en el final, añadiendo además una comprobación de colisiones.

En caso de tener que implementar un algoritmo (no implementado en la librería OMPL), se habría desarrollado el diagrama de Voronoi, puesto que su objetivo principal es alejarse la mayor distancia posible de los obstáculos. Además, este método elige siempre el camino más despejado, por lo que es favorable en entornos

donde hay abundantes obstáculos, como sería el caso de la mina. Este aspecto ha de ser tenido en consideración pues existe la posibilidad de que los túneles dentro de la mina sean estrechos y tengan numerosos obstáculos. Es por ello, que cuanto más alejado se mantenga el submarino de estos objetos, menor será la posibilidad de que colisione con alguno de ellos; suponiendo así un incremento del coste del proyecto debido a las reparaciones necesarias y los dispositivos necesarios para rescatar al submarino que realizaba la misión.

Se ha de destacar que se han cumplido todos los objetivos propuestos en la sección 1.3 de esta memoria, destacando los siguientes:

- El estudio y comprensión de los distintos métodos y algoritmos para la planificación de trayectorias.
- El aprendizaje y la comprensión del sistema operativo más potente de robótica *ROS* y de varias de sus herramientas.
- La generación de una trayectoria con el Girona 500.

El proyecto ha permitido adquirir y aplicar conocimientos ya adquiridos a lo largo de toda la carrera, además de afianzar el área de la ingeniería al que dedicar los próximos años de la carrera. Mientras que ya había una sólida formación en campos como la programación, la robótica y su aplicación industrial, se ha tenido la oportunidad de obtener conocimientos más específicos de los que no se tenía apenas noción, como puede ser en los numerosos algoritmos empleados en la planificación de trayectorias, o en entornos de programación y las comunicaciones en su interior, como se ha aprendido con *ROS*.

9.2. Trabajos futuros

Este trabajo realizado podría dar lugar a otros trabajos como la conexión entre *UWSim* y la librería *MoveIt*, para poder probar las trayectorias en el propio simulador e incluir la dinámica en posteriormente. A continuación de esto, se podría generar la conexión entre el simulador en sí, *UWSim*, y el robot real, pudiendo probar así todas las trayectorias generadas en el entorno real.

10. Presupuesto y planificación

En esta sección se muestra la planificación que se ha seguido para desarrollar el proyecto en su totalidad. Para ello se expondrán las fases de desarrollo, se indicará el esfuerzo en cantidad de horas dedicadas por sección y por último se mostrará la planificación detalladamente empleando un diagrama de Gantt.

10.1. Planificación

10.1.1. Fases de Desarrollo

El trabajo se encuentra dividido en 4 fases diferentes, en función de la tarea a realizar.

1. **Primera fase: Formación.** En esta parte, se ha realizado una comprensión del proyecto, se ha investigado sobre la planificación de trayectorias, aprendizaje del entorno que se empleará y de las herramientas utilizadas; además del estudio de los diferentes algoritmos para la planificación de trayectorias.

2. **Segunda fase: Preparación del entorno.** En primer lugar se prepara el sistema operativo en el que se va a trabajar (Ubuntu 14.04); a continuación, se instala *ROS*, *UWSim*, *MoveIt* y *Rviz*, que son los simuladores con los que se va a trabajar.

3. **Tercera fase: Desarrollo.** Generación de los ficheros necesarios tanto del robot como del entorno para poder generar las trayectorias.

4. **Cuarta fase: Elaboración de la memoria.** Recopilación y organización de la información, además de la documentación del proyecto.

10.1.2. Horas dedicadas

Una vez que las fases de desarrollo están correctamente definidas, se ha de evaluar cuantas horas se han dedicado a cada una de ellas, pudiendo dimensionar el esfuerzo realizado y su distribución del mismo en función de la fase.

A continuación, en la tabla 10, se muestra el número de horas dedicadas por fase aproximadamente.

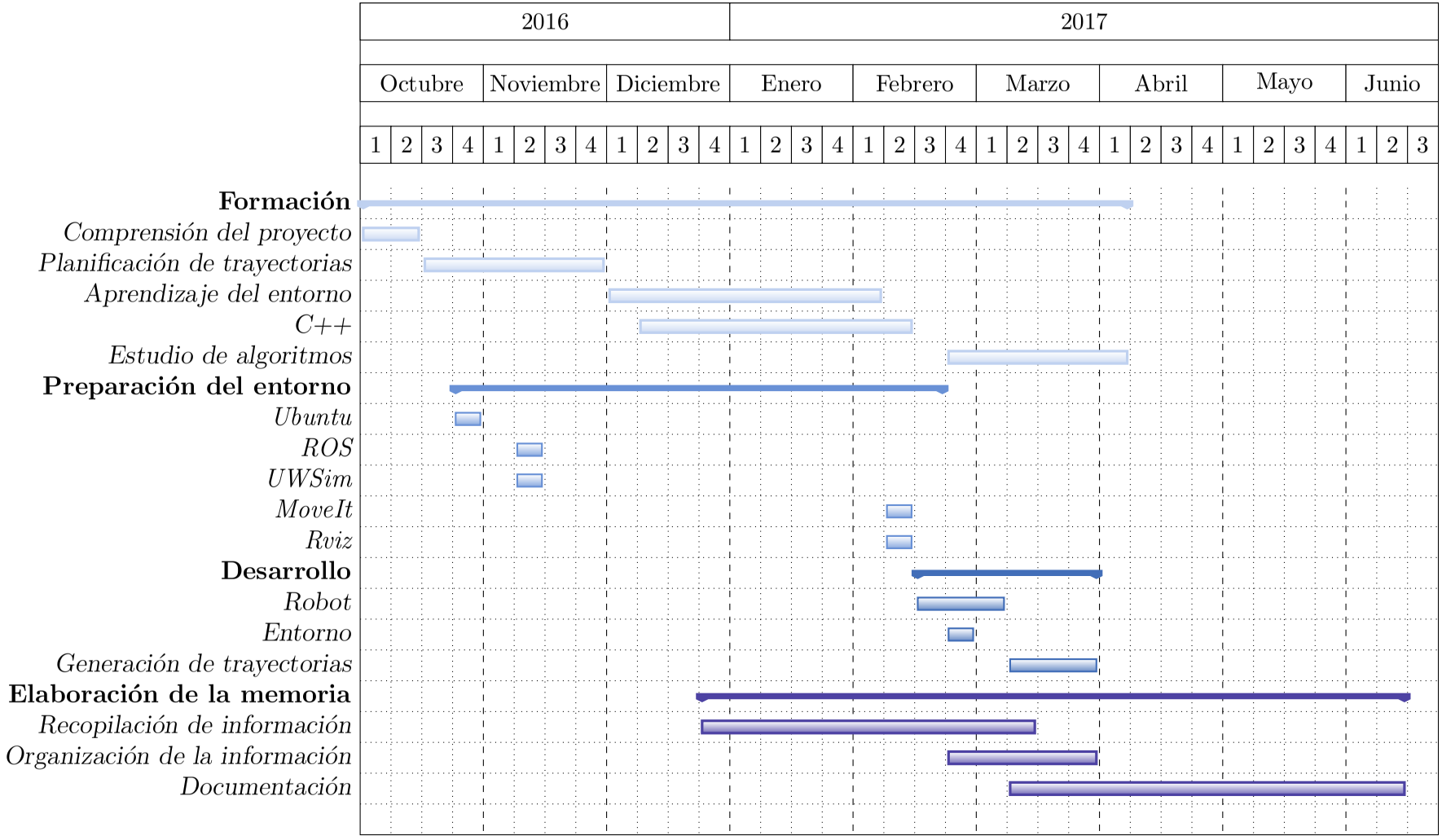
Fase	Horas
Formación	160
Preparación del entorno	40
Desarrollo	40
Elaboración de la memoria	160
TOTAL	**400**

Tabla 10: Horas empleadas

El número de horas totales dedicadas al trabajo es la suma de las horas dedicadas por cada una de las fases.

10.1.3. Diagrama de Gantt

En esta sección se muestra un diagrama de Gantt con la planificación desglosada por fases y actividades.


2016
2017
Octubre
Noviembre
Diciembre
Enero
Febrero
Marzo
Abril
Mayo
Junio
1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3
Formación
Comprensión del proyecto
Planificación de trayectorias
Aprendizaje del entorno
C++
Estudio de algoritmos
Preparación del entorno
Ubuntu
ROS
UWSim
MoveIt
Rviz
Desarrollo
Robot
Entorno
Generación de trayectorias
Elaboración de la memoria
Recopilación de información
Organización de la información
Documentación

10.2. Presupuesto

10.2.1. Costes de Ejecución

Es esta sección se desglosan los costes asociados al trabajo, quedando así una separación de hardware, software y personal.

10.2.1.1 Costes de personal

Teniendo en cuenta el número de horas empleadas en realizar este proyecto, se pueden derivar los costes de personal.

Personal	Cantidad horas	Precio Ud. (€)	Precio Total (€)
Ingeniero junior	400	20	8.000
TOTAL			**8.000**

Tabla 11: Costes de Personal

10.2.1.2 Costes de hardware

Hardware	Cantidad	Coste (€)	%Uso	Meses empleado	Amort (años)	Precio Total (€)
Portátil HP Pavilion	1	650	50	9	4	60,94
Disco duro 500 Gb	1	50	100	9	4	9,37
TOTAL						**70,31**

Tabla 12: Costes de Hardware

10.2.1.3 Costes de software

Los costes por software son los relacionados con los programas que han sido utilizados para realizar este proyecto. Algunas aplicaciones como Solid Edge en alguna de sus versiones es de coste, no obstante se ha utilizado la versión de estudiante que es gratuita y tenía las herramientas necesarias para realizar el diseño 3D necesario.

Software	Cantidad	Precio Ud. (€)	Precio Total (€)
Licencia *Ubuntu*	1	0	0
Licencia *ROS*	1	0	0
Licencia *UWSim*	1	0	0
Licencia *MoveIt*	1	0	0
Licencia *Rviz*	1	0	0
Licencia *Solid Edge*	1	0	0
Licencia *SketchUp*	1	0	0
Licencia *Blender*	1	0	0
TOTAL			**0**

Tabla 13: Costes de Software

10.2.2. Importe total

Costes	Precio Total (€)
Costes de Hardware	70,31
Costes de Software	0
Costes de Personal	8.000
TOTAL	**8.070,31**

Tabla 14: Coste total

Referencias

[1] Prats, M.; Perez, J.; Fernandez, J.J.; Sanz, P.J.: *An open source tool for simulation and supervision of underwater intervention missions*, 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 2577-2582, 7-12 Oct. 2012

[2] http://www.ijser.org/researchpaper/A-Review-Paper-on-Autonomous-Underwater-Vehicle.pdf

[3] D. Richard Blidberg: *The Development of Autonomous Underwater Vehicles (AUV).* Proyecto de investigación del Autonomous Undersea Systems Institute, Lee New Hampshire, USA

[4] http://www.auvsi.org/home/learnmore

[5] http://auvac.org/configurations/view/155

[6] http://www.underwaterwork.es/division-rov.html

[7] Antonio Galo Ruiz Ortega: *Diseño de un Sistema autónomo para un Vehículo Submarino* Proyecto fin de Carrera. Ingeniería Electrónica. Universidad Politécnica de Cataluña.

[8] http://www.nuestromar.org/noticias/categorias/63394-08-16/crece-inter-s-veh-culos-submarinos-aut-nomos-explotaciones-petroleras-en-mar

[9] https://www.cepsa.com/Informe_Anual_y_de_Responsabilidad_Corporativa_2014/pdf/CEPSA_IA_2014_ESP.pdf pág.116-122.

[10] http://www-assig.fib.upc.es/ñob/protegit/treballs/Q2_03-04/submarinos/rov_clases.htm

[11] http://laplace.us.es/wiki/index.php/ %C3 %81ngulos_de_Euler_y_de _navegaci %C3 %B3n

[12] http://catarina.udlap.mx/u_dl_a/tales/documentos/msp/de_l_ea/capitulo1.pdf

[13] http://www.navaldrones.com/SPURV.html

[14] https://www.eu-robotics.net/sparc/newsroom/sparc-in-the-press/robotics-and-society-discussion-at-the-french-parliament.html?changelang=2

[15] http://wiki.ros.org/rviz

[16] http://www.irs.uji.es/rauvi/news.html

[17] http://www.irs.uji.es/trident/

[18] https://trid.trb.org/view.aspx?id=8527

[19] https://books.google.es/books?id=prjMtIr_yT8C&pg=PA427&lpg=PA427
&dq=EAVE+Vehicle+at+the+Marine+Systems+Engineering+Lab&source=bl&ots=-
1U6kxT8mq&sig=4byh WOa0Gqu4HmDuXY37SBSnQ6s&hl=es&sa=X&ved=0ahUKEwjRo5j5
page&q=EAVE %20Vehicle %20at %20the %20Marine %20Systems %20Engineering %20Lab&f=

[20] http://www.afcea.org/content/?q=draper-laboratory-support-autonomous-undersea-vehicles

[21] http://www.ieeeoes.org/history/080523-010.pdf

[22] http://www.cs.mun.ca/ãv/papers/cook14simulators.pdf

[23] http://wiki.ros.org/gazebo

[24] http://robotcamixo.blogspot.com.es/2010/12/aplicaciones-de-los-robots.html

[25] https://es.scribd.com/document/260888962/AUV-history

[26] http://tonicarpio.com/despieces/misiones-para-los-auv

[27] García García J. J.: *Desarrollo de una Herramienta Informática para la Simulación Dinámica de Vehículos Submarinos No Tripulados.* Proyecto fin de carrera, Universidad Politécnica de Cartagena, 2013.

[28] Lentin Josseph: *Mastering ROS for Robotics Programming* 1st ed. Birmingham: Packt Publishing, 2015. Print.

[29] http://moveit.ros.org/documentation/

[30] http://wiki.ros.org/ROS/Tutorials

[31] http://wiki.ros.osg/ROS/Concepts

[32] http://www.ros.org/

[33] http://www.ros.org/about-ros/

[34] Morgan Quigley, Brian Gerkey & William D. Smart. *Programming robots with ROS.* O'Reilly Media 2013.

[35] Morgan Quigley, Brian Gerkey & William D. Smart: *Programming robots with ROS.* O'Reilly Media 2013.

[36] http://ompl.kavrakilab.org/planners.html

[37] J.-c.: *Latombe, Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[38] H.Choset, K. M. Lynch, S. Hutchinson, G. Kantos, W.Burgard, L. E. Kavrati, and S.Thrun: *Principles of Robot Motion: Theory, Algorithms, and Implementations.* MIT Press, 2005.

[39] N. M. Amato and G. Song: *"Using motion planning to study protein folding pathways", Journal of Computational Biology.* April 2002.

[40] S. LaValle: *"Motion planning" Robotics and Automation Magazine, IEEE (Volume:18 , Issue: 1)*, 2011.

[41] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein: *Introduction to Algorithms.* 2 ed.,2001.

[42] P. Hart, N. Nilsson, and B. Raphael: *"A formal basis for the heuristic determination of minimum cost paths", Systems Science and Cybernetics, IEEE Transactions on, vol. 4*, pp. 100–107, July 1968.

[43] https://www.plm.automation.siemens.com/en_us/products/solid-edge/index.shtml

[44] https://www.blender.org/

[45] http://www.irs.uji.es/uwsim/wiki/index.php?title=Main_Page

[46] http://www.irs.uji.es/uwsim/wiki/index.php?title=Configuring_and_creating_scenes

[47] http://cirs.udg.edu/auvs-technology/auvs/

[48] Maxim Likhache: *SBPL, Search Based Planning Library.* 2009. url: http://www.sbpl.net

[49] Ioan A. Şucan, Mark Moll y Lydia E. Kavraki: *"The Open Motion Planning Library"* En: IE*EE Robotics & Automation Magazine 19.4* (dic de 2012)

[50] Maxim Likhachev, Geoffrey J Gordon y Sebastian Thrun: *"ARA*: Anytime A* with provable bounds on sub-optimality"* En: *Advances in Neural Information Processing.* Systems. 2003, None

[51] Maxim Likhachev y col: *"Anytime Dynamic A*: An Anytime, Replanning Algorithm."* En: *Proceedings of the International Conference on Automated Planning and Scheduling* (ICAPS). Jun. de 2005, págs. 262-271.

[52] Maxim Likhachev y Anthony Stentz: *"R* search"*. En: *Lab Papers* (GRASP) (2008), pág. 23

[53] Lydia E Kavraki y col. *"Probabilistic roadmaps for path planning in high-dimensional configuration spaces"*. En: *IEEE Transactions on Robotics and Automation 12.4* (1996), págs. 566-580.

[54] Sertac Karaman y Emilio Frazzoli: *"Sampling-based algorithms for optimal motion planning"*. En: *The International Journal of Robotics Research 30.7* (2011), págs. 846-894.

[55] Robert Bohlin y Lydia E Kavraki: *"Path planning using lazy PRM"* En: *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*. Vol. 1. IEEE. 2000, págs. 521-528.

[56] Andrew Dobson, Athanasios Krontiris y Kostas E Bekris: *"Sparse roadmap spanners"*. En: *Algorithmic Foundations of Robotics X*. Springer, 2013, págs. 279-296

[57] Andrew Dobson y Kostas E Bekris: *"Improving sparse roadmap spanners"*, En: *Robotics and Automation (ICRA)*, 2013 IEEE International Conference on. IEEE. 2013, págs. 4106-4111

[58] Steven M LaValle *Rapidly-Exploring Random Trees A New Tool for Path Planning*. Inf. téc. Computer Science Dept., Iowa State University, oct. de 1998.

[59] James J Kuffner y Steven M LaValle. *"RRT-connect: An efficient approach to singlequery path planning"* En: *Robotics and Automation*, 2000. Proceedings. ICRA'00. IEEE International Conference on. Vol. 2. IEEE. 2000, págs. 995-1001.

[60] Robert Bohlin y Lydia E Kavraki. *"A Randomized Approach to Robot Path Planning Based on Lazy Evaluation"* En: *COMBINATORIAL OPTIMIZATION-DORDRECHT- 9.1* (2001), págs. 221-249.

[61] Sertac Karaman y Emilio Frazzoli. *"Sampling-based algorithms for optimal motion planning"* En: *The International Journal of Robotics Research 30.7* (2011), págs. 846-894.

[62] Oren Salzman y Dan Halperin *"Asymptotically near-optimal RRT for fast, high quality, motion planning"* En: *Robotics and Automation (ICRA)*, 2014 IEEE International Conference on. IEEE. 2014, págs. 4680-4685

[63] Léonard Jaillet, Juan Cortés y Thierry Siméon. *"Sampling-based path planning on configuration-space costmaps"*, En: *Robotics, IEEE Transactions on 26.4* (2010), págs. 635-646.

[64] David Hsu, Jean-Claude Latombe y Rajeev Motwani. *"Path planning in expansive configuration spaces"*. En: *Robotics and Automation, 1997. Proceedings.*, 1997 IEEE International Conference on. Vol. 3. IEEE. 1997, págs. 2719-2726.

[65] Gildardo Sánchez y Jean-Claude Latombe. *"A single-query bi-directional probabilistic roadmap planner with lazy collision checking*. En: *Robotics Research*. Springer, 2003, págs. 403-417.

[66] Ioan A Şucan y Lydia E Kavraki *Kinodynamic motion planning by interior-exterior cell exploration"*. En: *Algorithmic Foundation of Robotics VIII*. Springer, 2010, págs. 449-464.

[67] Bryant Gipson, Maciej Moll y Lydia E Kavraki. *"Resolution independent density estimation for motion planning in high-dimensional spaces"*. En: *Robotics and Automation (ICRA)*, 2013 IEEE International Conference on. IEEE. 2013, págs. 2437-2443.

[68] Ian Alon Francisco Yon Yon: *Algoritmo para manipulación de objetos en un robot PR2*. Proyecto de fin de carrera, Universidad de Chile, 2016.

[69] http://www.sketchup.com/es

[70] http://www.spanish-translator-services.com/espanol/t/infoset.htm

[71] http://wiki.ros.org/urdf

[72] http://moveit.ros.org/install/

[73] http://www.irs.uji.es/uwsim/wiki/index.php?title=Installing_UWSim

[74] http://moveit.ros.org/documentation/planners/

[75] http://www.openthefile.net/es/extension/dae

[76] http://www.garrigues.com/es_ES/noticia/la-regulacion-de-los-drones-marinos

[77] http://www.faqs.org/faqs/windows-emulation/wine-faq/

[78] Latombe, J C *Robot Motion Planning*, Kluwer Academic Pulisher, 1991.

[79] Muñoz, V. *Planificación de Trayectorias para Robots Móviles*, Tesis Doctoral. Universidad de Málaga,1995.

[80] http://rabida.uhu.es/dspace/bitstream/handle/10272/5501/Nuevas_aportaciones_en_algoritmos_de_planificacion.pdf?sequence=2

[81] https://www.ijser.org/researchpaper/A-Review-Paper-on-Autonomous-Underwater-Vehicle.pdf

[82] https://www.researchgate.net/figure/261267388_fig6_Figura-8-a-Escena-Real-b-Trayectoria-Encontrada-entre-Punto-Origen-rojo-y-Destino

[83] Chazelle, B.: *Aproximation and Decomposition of Shapes*, in Algorithmic and Geometric Aspects of Robotics, Lawrence Erlbaum Associates, Hillsdale, NJ, pags. 145-185.

[84] Aurenhammer, F: *Voronoi diagrams: A survey of fundamental geometric data structure*, ACM Comput. Surv.,nº 23: pags 345-405

[85] Wilmarth, S.A., Amato N.M., Stiller, P.F.: *"A probabilistic roadmap planner with Sampling on the medial Axis of the Free Space*, TR-Department of Computer Science Texas A&M University, 1998.

[86] A. Ollero: *Planificación de trayectorias para Robots Móviles*, Universidad de Málaga, 1995

[87] https://ec.europa.eu/digital-single-market/en/policies/robotics

[88] http://vicorob.udg.edu/portfolio_page/comarob/

[89] http://cirs.udg.edu/portfolio_page/comarob/

[90] Choset, H.: *Principles of Robot Motion*, The MIT Press, 2016

[91] Sethian, J.A. *A fast marching level set method for monotonically advancing fronts*, Proc. Natl. Acad. Sci. USA Vol. 93, pp. 1591-1595. Applied Mathematics. 1996

A. Anexo I. Manual de herramientas

A.1. Instalación de *ROS*

Una vez explicados todos los conceptos trabajados a lo largo del proyecto, se procede a explicar como ha sido instalado *ROS*.

Existen distintas versiones de *ROS* en función del año en el que han sido emitidas, ya que el objetivo es conseguir la versión más estable en función del robot y sus necesidades.

Para este proyecto se ha utilizado la versión *Indigo*, instalada bajo el sistema operativo de *Ubuntu* en la versión 14.04.. *ROS* tiene una plataforma en la cual hay tutoriales que indican paso a paso como instalar cualquiera de sus versiones en cualquier sistema operativo. En este caso se han seguido las instrucciones para la instalación en *Ubuntu*.

Al igual que para el resto de aplicaciones en *Ubuntu*, la instalación de *ROS Indigo* se produce escribiendo el siguiente código en la consola.

```
sudo apt-get install ros-indigo-destop-full
```

A la hora de trabajar con *ROS*, se ha de crear en primer lugar un espacio de trabajo ("*workspace*") donde se irán instalando los sucesivos paquetes. El nombre más intuitivo que se le asigna a esta carpeta es *cat_ws*. El *workspace* se crea empleando la herramienta *catkin_make*; dentro de esta carpeta hay otras tres que son *build*, *devel* y *src*; siendo esta última donde se instalarán los paquetes que se van a utilizar o crear.

Como se puede observar en la imagen 60, dentro de la carpeta *src* se encuentran todos los paquetes de *ROS* que se han instalado.

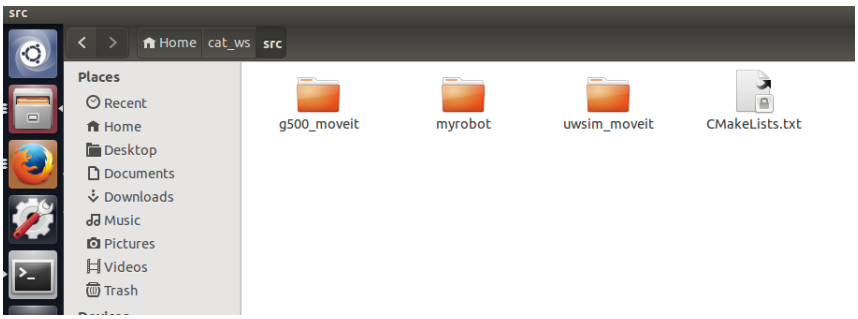


Figura 60: Carpeta *src* dentro de *cat_ws*

En la imagen siguiente se muestra la estructura interna del paquete que se va a crear para este proyecto, siendo similar a la estructura de la mayoría de los paquetes. Como se ha indicado, los paquetes pueden tener en su interior toda la información que se considere relevante para el susodicho paquete, pero siempre tendrán un *CMakeLists.txt* ya que en él se incluye la información que permite construir el paquete, y el manifiesto, que es el archivo que provee de toda la información necesaria acerca del paquete.

En este caso el paquete ha sido creado por el asistente *MoveIt* y se han creado dos carpetas *config* y *launch* (como se puede observar en la imagen 61. En la primera de estas carpetas se encuentran las descripciones del robot y todas las características establecidas con el asistente referentes al submarino; mientras que en la segunda se encuentran los archivos que se podrán ejecutar.

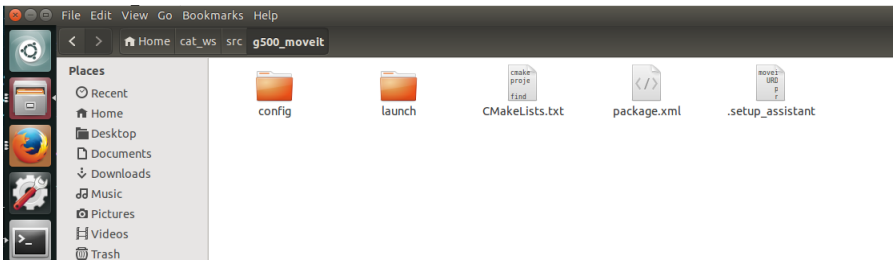


Figura 61: Paquete de *ROS*

A.2. Instalación de *MoveIt*

Una vez que ya se ha instalado *ROS Indigo*, se puede proceder a la instalación del resto de herramientas; en este caso se explicarán las instrucciones para instalar *MoveIt*.

Para proceder a la instalación se han seguido los pasos de página oficial de MoveIt [72]. La instalación de este paquete es muy sencilla, puesto que únicamente hay que lanzar un comando en la terminal y a continuación seguir las instrucciones que se muestran en ella:

```
sudo apt-get install ros-indigo-moveit
```

Para aprender a utilizar este paquete se descargó el ejercicio de prueba *PR2* con el siguiente comando en la consola:

```
sudo apt-get install ros-indigo-moveit-full-pr2
```

Por último, para instalar el entorno, es necesario lanzar la siguiente instrucción:

```
source /opt/ros/indigo/setup.bash
```

Una vez que ya se ha finalizado la instalación, se puede proceder a la realización de los tutoriales.

A.3. Instalación de *UWSim*

Dependiendo de los requerimientos y del sistema existen varias opciones de instalación.

- **Package-base installation:** esta instalación es la mas sencilla de todas, pero carece de las últimas características.

- **UWSim Source-based installation:** instala el simulador desde la fuente y se descarga las librerías de los paquetes de *UWSim*. Esta instalación permite una instalación rápida y contiene las últimas versiones, no obstante no se pueden modificar las librerías con esta instalación.

- **Full Source-based installation:** es la instalación más completa. Instala el simulador y las librerías desde la fuente. Hay un control completo de desarrollo, las últimas características, pero la instalación es más complicada y lenta.

En primer lugar, para familiarizarse con el simulador y el entorno de *ROS* se realizó la instalación más sencilla, es decir, *Package-base installation*. Para instalarlo únicamente había que introducir en la consola el comando [73]:

```
sudo apt-get install ros-indigo-uwsim
```

Conforme transcurrió el tiempo y se fue cogiendo soltura con el simulador, como no se sabían todas las funciones que serían necesarias se instaló la versión completa que permitía modificar cualquier aspecto del simulador, ya fuese el robot o el entorno.

Para realizar esta instalación se siguieron las instrucciones de la *Wiki* [73] (página oficial de *ROS*).

1. Crear un workspace de *ROS*. El workspace de *ROS* es una carpeta donde se encuentran todos los fuentes de nuestro proyecto en una estructura lógica. Para crearlo se introducen los siguientes comandos en la terminal:

```
mkdir -p /uwsim_ws/src
```

```
cd /uwsim_ws/src
```

```
catkin_init_workspace
```

```
mcd /uwsim_ws
```

```
catkin_make install
```

2. Añadir el *workspace* a nuestro bashrc. Si se añade el *workspace* creado al bashrc, cada vez que se arranque la consola, estará disponible esta variable de entorno y se podrá ejecutar sin problema los programas. Para ello se edita el fichero " */.bashrcz* se añade al final:

```
source /uwsim_ws/install/setup.bash
```

3. Instalar librerías para UWSim.

```
sudo apt-get install ros-indigo-uwsim-bullet ros-indigo-uwsim-osgbullet ros-indigo-uwsim-osgocean ros-indigo-uwsim-osgworks ros-indigo-visualization-osg
```

4. Definir las fuentes desde donde se instalará UWSim.

```
cd /uwsim_ws/
```

```
gedit .rosinstall
```

Se rellena el fichero con lo siguiente:

```
- other: {local-name: /opt/ros/indigo/share/ros}
- other: {local-name: /opt/ros/indigo/share}
- other: {local-name: /opt/ros/indigo/stacks}
- setup-file: {local-name: /opt/ros/indigo/setup.sh}
- git: {local-name: src/uwsim_osgocean,
    uri: 'https://github.com/uji-ros-pkg/uwsim_osgocean.git',
version: indigo-devel}
- git: {local-name: src/uwsim_osgworks,
    uri: 'https://github.com/uji-ros-pkg/uwsim_osgworks.git',
version: indigo-devel}
- git: {local-name: src/uwsim_bullet,
      uri: 'https://github.com/uji-ros-pkg/uwsim_bullet.git',
version: indigo-devel}
- git: {local-name: src/uwsim_osgbullet,
    uri: 'https://github.com/uji-ros-pkg/uwsim_osgbullet.git',
version: indigo-devel}
- git: {local-name: src/underwater_simulation,
    uri: 'https://github.com/uji-ros-pkg/underwater_
simulation.git', version: indigo-devel}
- git: {local-name: src/visualization_osg,
    uri: 'https://github.com/uji-ros-pkg/
visualization_osg.git', version: indigo-devel}
```

5. Descargar ficheros

```
rows update
```

6. Instalar dependencias y compilar

```
rosdep install –from-paths src –ignore-src –rosdistro indigo -y
```

```
catkin_make install
```

Una vez instalado todo el sistema y compilado, se ejecutará por primera vez la escena predefinida de *UWSim* y con ello se comprobará que está bien definido.

1. Asegurarse de que *roscore* está en marcha. Este es un servidor que se encarga de la comunicación entre procesos de *ROS*, y por tanto es necesario que esté funcionando en todo momento. Para ello se abrirá una terminal nueva y ejecutar:

```
roscore
```

2. Ejecutar *UWSim*

```
rosrun uwsim uwsim
```

   La primera vez que se ejecute *UWSim* se pedirá descargar un fichero de datos, se ha de aceptar para que se descargue los modelos básicos de terrenos, vehículos, etc. que se instalarán en */.uwsim*.

3. Comprobar que *UWSim* está publicando información. *UWSim* utiliza las interfaces de *ROS* para simular sensores como cámaras y sonares entre otros. Se puede ver todo lo que se está publicando en una nueva terminal con el siguiente comando:

```
rostopic list
```

A.4. Instalación de *Rviz*

Instalar *Rviz* es sencillo en comparación con otras instalaciones llevadas a cabo para este proyecto. Únicamente habrá que introducir en la terminal los siguientes comandos:

```
sudo apt-get update
```

```
sudo apt-get install ros-indigo-rviz
```

Una vez instalado es necesario introducir el siguiente comando en la terminal:

```
source /opt/ros/indigo/setup.bash
```

B. Anexo II. Acrónimos

ACM	Allowed Collision Matrix
AHRS	Attitude and Heading Reference System
AOSN	Autonomous Ocean Sampling Network
ARA*	Anytime Repairinig A*
AUV	Autonomous Underwater Vehicle
BKPIECE	Bi-directional KPIECE
CAD	Computer Aided Design
CAM	Computer Aided Manufacturing
CHOMP	Covariant Hamiltonian Optimization for Motion Planning
CIRS	Centro de Investigación en Robótica Submarina
COLLADA	COLLAborative Design Activity
CRG	Cilindros Rectilíneos Generalizados
DNS	Domain Name System
DVL	Doppler Velocity Log
EST	Expansive Spaces Trees
FCL	Full Container Load
GPS	Global Positioning System
GUI	Graphical User Interface
IMU	Inertial Measurement Unit
KPIECE	Kinematic Planning by Interior-Exterior Cell Exploration

LBKPIECE	Lazy Bi-directional KPIECE
LBTRRT	Lower Bound Tree RRT
OMPL	Open Motion Planning Library
OSG	OpenSceneGraph
PDST	Path-Directioned Subdivision Trees
PI^2	Policy Improvement with Path Integrals
PRM	Probabilistic RoadMaps
pRRT	Parallel RRT
pSBL	Passive Spam Block List
ROS	Robot Operating System
ROV	Remotely Operated Robot
RRT	Rapidly-exploring Random Trees
SBL	Single-query Bi-directional Lazy collision cheing planner
SBPL	Searched-Based Planning Library
SRDF	Semantic Robot Description Format
STL	Standard Triangle Language
STOMP	Stochastic Trajectory Optimization for Motion Planning
STRIDE	Search Tree with Resolution Independent Density Estimation
T-RRT	TRansition-based RRT
UARS	Unmanned Artic Research Submersible
URDF	Unified Robot Description Format

USBL	Ultra-Short BaseLine
UWSim	UnderWater Simulator
VCS	Version Control System
ViCOROB	Instituto de VIsión por COmputador y ROBótica
XML	eXtensible Markup Language